5-1-1993

Design of a hardware efficient key generation algorithm with a VHDL implementation

James A. Goeke

Design of a Hardware Efficient Key Generation Algorithm with a VHDL Implementation

by

James A. Goeke

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Graduate Advisor - Prof. George A. Brown

Department Chairman - Dr. Roy Czernikowski

Reader - Dr. Tony Chang

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 1993

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title of Thesis: Design of a Hardware Efficient Key Generation Algorithm with a VHDL Implementation.

I, James A. Goeke, hereby deny permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Date: 19 May 93

This document was produced using Windows version 3.1, Wordperfect version 5.2 for Windows, Windows Paintbrush, and Mathcad version 3.1 for Windows. The C programs were developed using Borland C version 3.1. All of these programs were run on a Gateway 2000 386/33 PC clone. The VHDL portions were developed on a Sun SparcStation 10. The code was written using the emacs editor, and a syntax directed editor extension available on the Internet. The analysis and simulation of the VHDL was done using Vantage Spreadsheet. The C code was also compiled and run on the Sun using CC, an ANSI compatible C compiler from Sun.

The following names used here and in the remainder of the document are registered trademarks of the respective companies:

Windows	Microsoft Corporation,
Wordperfect	Wordperfect Corporation,
Paintbrush	Microsoft Corporation,
Mathcad	Mathsoft Corporation,
Borland	Borland International,
Sun	Sun Microsystems,
Vantage Spreadsheet	Vantage Analysis Systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES AND EQUATIONS

ABSTRACT

The design and implementation of a key-generation algorithm will be discussed. The steps in the procedure consist of choosing a baseline algorithm for comparison, designing a new algorithm, testing and comparing the algorithms using C language programs, and then implementing the new algorithm using VHDL. The final result of testing the algorithm implemented in VHDL will be compared to the original results obtained from the C program implementing the new algorithm.

C language programs of the chosen algorithms will be developed to verify their similarity and functionality. The results will be used to decide if the new algorithm selected for implementation is sufficiently robust, and similar in functionality to the baseline algorithm.

After the algorithm is selected, it will be implemented in a hardware description language. This will be done for purposes of demonstrating top down design and hardware efficiency. This process will involve the steps of moving the design from an abstract algorithmic view, to a high level (behavioral) hardware description, and then to a low level (structural) description. This process will illustrate the details of moving the design from a theoretical level to a practical implementation level.

1.0 INTRODUCTION

Processors of all sizes, from single board microprocessors to large main frames, can perform many complex and varied tasks. But they all suffer from a common problem. No matter how large or complex the system, it has a finite amount of working storage space, and usually a much smaller amount of space to actually perform the processing. The amount of storage space available is increasing dramatically through the use of new technologies, but so are the size of the data sets that need to be processed.

If the problem to be solved requires more data than the amount of available storage, this is a serious fault. But often the data can be more compactly stored because it only sparsely occupies the storage space it is using. Key generation algorithms can be used to perform this memory space compaction.

Another aspect of these data sets is that they must often be accessed based on the new input to the program. As the new data arrives, it must either be stored, compared to something already in storage, used to change already stored data, etc. There are many varied methods to both store the data and to perform the accesses, but they all depend on being able to uniquely associate the data with a storage location or locations. As with the storage limitations, this brings up another shortcoming of processors. No matter how big or how fast the processor is (and again this is rapidly improving with new technologies), if the data set is large enough, it will eventually cause very poor performance. Key generation algorithms play a very important part in processing large data sets since they either determine which part

of the incoming data to use, or somehow generate a new key from the incoming data. If the key is generated properly, it can potentially allow a more direct access to the desired storage location, and thus improve performance by allowing the processor to work on the problem rather than just using its power to search its own memory.

Key generation algorithms are an important part of many larger algorithms. They provide an important device to allow storage of potentially large volumes of data in a much smaller memory space, and they provide a method of associating incoming data with locations within the processors storage space. Because of this, key generation algorithms have been widely studied. But most of this effort has been on how to generate keys using software. As technology advances, many of the algorithms that historically were done in software are now being done in hardware. This presents a problem for key generation algorithms. Many of the current algorithms are based on ideas that do not translate well into hardware, or are very inefficient in hardware. Because of this, it is my desire to work on the development of a key generation algorithm that will be comparable in performance to a currently used software algorithm, and at the same time be designed with hardware efficiencies and limitations in mind.

2.0 CONCEPT

There are many methods used to associate, or key, data to a storage location. Some of the possibilities include using: a particular element in the data (e.g. name), a set or combination of elements in the data (e.g. name and address), a number assigned to each piece of data, a linking of the data together in a list, etc. No matter how the data is referenced to a location, it must have the property of having a one-to-one correspondence between the attributes used as the key and the location

The purpose of this work is not to examine key generation algorithms in detail. Rather, it is to examine how to build an efficient one in hardware. To select a baseline algorithm, an informal survey of software examples was examined, and discussions were held with a number of people involved in the design of both hardware and software. The result of this was the conclusion that hashing is one of the most commonly used key generation algorithms. It was decided to use this as the baseline algorithm for comparison.

Having decided on hashing, it should be noted that this violates the last property mentioned in the first paragraph, namely, that the key and the storage location have a one-to-one correspondence. This limitation can be overcome by developing the algorithm to compensate for these collisions. There are a large number of methods to do this that place the data in an alternate storage location. The main requirement is that they be deterministic so that the same input will result in the same collision, and then the same alternate location.

Since the collision strategy often involves knowledge of the program and/or system using it, it is beyond the scope of this work.

Hashing is a type of key generation that gets its name because it hashes, or scrambles the bits of the piece of data to be used as the key. This is a widely used technique because it is fast, simple to implement in software, can use minimal storage space, and can take the search directly to the desired storage location (as opposed to a linear search). Repeated use and testing have demonstrated this to be one of the most simple and robust methods. It is due to the power and simplicity of hashing that it is so widely used, even thought it has the drawback of producing collisions.

In addition to having a key generation algorithm chosen, some type of larger algorithm or process is needed to use the functionality of the key generation algorithm. For this work, the skeleton of the type of algorithm that could be used in a data searching or data compression application will be used. It was selected because it will provide a very simple framework within which to use the key generation algorithm while at the same time providing a more realistic test.

The skeleton algorithm will operate by using the incoming data, and another value that will represent the history of some section of the data that has come immediately before the current piece of data. A key will be generated using hashing to represent this entire quantity of data. The key will be used as the reference into storage. The operations that result from the values found in storage will determine if the algorithm compresses the data, signifies a match (or mismatch) to previous data results, etc. These activities are not of importance to

the scope of this work. One element that the skeleton will reference from storage will be a value assumed to be stored there that will represent the new value for the history of the data. This new history value will be used with the next piece of input data.

2.1 SOFTWARE ALGORITHM

The actual skeleton operation will be as follows. The incoming piece of data, which will be called the symbol, will be read in. This will be concatenated with the value representing the data history, which will be called the previous code. The result will be called the hash argument, or harg. The modulo of the harg will then be taken. This result will be the location in storage to be referenced, called the address (see Equation 1). An

modulo algorithm address=[(*symbol*) catanate (*previous code*)]mod 4021

Equation 1

actual algorithm would probably have several pieces of information stored there for its use. In this work, all that will be stored there will be a value that will become the new previous code, which will be combined with the next symbol. This process will then continue as long as there are more symbols to be processed.

The previous paragraph mentions that a modulo operation will be used. The choice of the modulus is an important parameter. The values most often used for this are prime numbers. While the modulus operation will produce good results, it is very difficult to implement in hardware. This is because a modulo is a type of division, and in hardware it

is difficult to perform division operations unless they are by a power of 2 (in which case it is merely a shift operation). Because the number of storage locations is most often picked to be to be some power of two (to make both hardware and software addressing easier) the modulus choosen will be a prime number smaller than the number of storage locations. This means that the operation will produce remainders that range from 0 to the one less than the prime number. One unfortunate side effect of this is that there will be some locations that will never be accessed. Any location that corresponds to a value from the address equal to the prime number up to the last address in the table will be a location that will not be able to be produced by the algorithm. This will not be serious if the prime number is chosen to be close in value to the number of storage locations. The prime number used in the selected hashing algorithm was picked from a table that listed a special value to use for many common table sizes (all of the table sizes were powers of 2). The table of selected primes used was from a piece of code developed by Kodak Berkley Research.

For this work, the actual processing that could take place outside of the hashing is not important, what is important is the hashing itself. For that reason, no activities occur in the skeleton except for the hashing. The information recorded will be how often a each particular address is accessed. This will be used as a measure of the efficiency of the hashing algorithms.

It was mentioned that values, called previous codes, would preexist at the storage locations. One reason for this is that they cannot be generated without developing a complete algorithm to surround the key generation algorithm. The generation of the previous codes is unique to and dependent on the type of processing being done. However, this is not an

unrealistic assumption. Some algorithms work by pre-computing values that are known to function well for a given type of data (this assumes that something is known about the type of data being input). Alternately, it could be assumed that a snapshot is being taken during the middle of a run, and the values at the locations were calculated earlier in the process and are now only being accessed.

2.2 HARDWARE ALGORITHM

Further investigation revealed that XOR operations were sometimes mentioned in relation to hashing. This was interesting because it is a very simple operation to perform in hardware. It is one of the "primitive" operations usually included in the set of logical operators in most programming languages and in many hardware devices. Because it is hardware efficient, the XOR operation made an excellent candidate for this work.

Unfortunately none of the references to using XOR operations for hashing gave an algorithm, they only mentioned that it could be used. But the underlying premise of hashing operations is that they will give an apparently random but uniform distribution of the data into the storage locations. It is highly desirable to avoid bunching or grouping of data as this will greatly increase the number of collisions.

After a great deal of investigation, looking at mappings and distributions, and running a large number of tests, one factor that seemed to have the greatest effect on producing a

better hashing was the number of bits being operated on. This observation, and further tests, lead to the development of the algorithm selected for hardware implementation.

The algorithm will mirror the procedure using the modulo. The skeleton of the algorithm will read in the data representing symbols. The symbol will be XOR'ed with a bit reversed representation of itself. This value will be concatenated with the previous code which will produce the harg. The address will be generated by shifting the harg down (toward the LSB) by the number of bits in the symbol and XOR'ing this with the previous code (see Equation 2). This substitute for the modulo at first glance seems much more

$$\textit{XOR algorithm address} = \frac{(\textit{symbol} \oplus \textit{reverse symbol}) \text{ catanate } \textit{previous code}}{256} \oplus \textit{previous code}$$

NOTE: division by 256 (which is 2^8) ≡ shift right by 8 bits

Equation 2

complicated. Further inspection reveals that each of the operations is actually very simple to implement in hardware. There are only two types of operations used, the XOR's, which are simple combinatorial operations, and the shift and bit reverse which are both just interconnect wiring that come for "free" in hardware. A benefit of this hashing function over the modulo is that it has the potential to use all the possible storage locations.

3.0 ALGORITHM ANALYSIS

To study the algorithms proposed in the first section, several C programs were written. A few of them will be discussed here because they relate to the generation and analysis of the data used to verify and compare the algorithms. The programs discussed are included in appendix A. Other programs were developed, but are not included as they helped develop the programs shown but do not add any additional information. The programs discussed here are: make-sym.c, histogra.c, smooth.c, and addr-gen.c.

Make-sym.c was one method used to generate the symbols and previous codes needed by the algorithms. It would have been possible to take any arbitrary file and use it as input data. This would have allowed generation of results, but the results would have been difficult to understand. Instead it was decided that maintaining control over the input data sets would allow more comprehensive interpretation of the results. One method of producing input was this C program. It was designed to generate data that had a random distribution. The reason for this was to check the algorithms against totally unstructured data. The source code and a brief description for this program are in appendix A1. The analysis of the data produced by this program is shown in Appendices B5 and B6.

In addition to the unstructured input data, additional data was generated using Mathcad on a personal computer. Mathcad allows the user to generate random values that fall into a distribution. Two distributions were chosen, a uniform and a gaussian distribution. These were used to check specific attributes of the algorithms. The uniform input was used to

check how the algorithm would respond to all input values being used with something approaching equality. The gaussian distribution was chosen for the opposite reason, to see the algorithm's response to inputs falling into a small range of the possible values.

Histogra.c was written to do high speed histogramming operations. It accepts as input either values representing symbols or values representing previous codes. It then generates counts that show how often each location in the range has been accessed. This function is available in Mathcad, but a stand alone program was much faster. The source code and a brief description for this program are in appendix A2.

The program smooth.c was written to perform simple one dimensional convolutions. It allows the user to specify the width over which the convolution is to be applied. To simplify the program, it only accepts odd values for the width so that there will always be a central value to convolve about. The actual convolution did an average over the entire width of the window. The source code and a brief description for this program are in appendix A3.

The final C program is addr-gen.c. This is the program that implemented the algorithm skeletons discussed in the CONCEPT section. The program read a file of previous codes and stored them in an array. As processing began, it read the symbols from another file, one symbol at a time. For each symbol read in, both the modulo and the XOR hashing algorithms were applied. Each algorithm generated an address that was used to increment a count indicating how often each particular address was generated. These counts were the products of this program. When it finished reading the input symbol file, it took the arrays

containing the number of accesses per location for both the modulo and XOR algorithms and wrote them out to separate files. The source code and a brief description for this program are in appendix A4.

3.1 HASHING

Hashing algorithms are not completely general purpose. They are designed for a specific size table. In this case it was decided to use a 4-kbyte table. This means that 12 bits are needed for addressing. This size was chosen as being large enough to be useful in a real algorithm, but small enough to keep the amount of data tractable. Since every address is associated with a previous code that means that 4096 12 bit codes had to be generated.

The input symbol size was chosen as 8 bits. This is a very standard value, and corresponds to the size of many buses and operations. It was chosen because it would be very likely that real world applications could use this symbol size. This meant that values in the range of 0 to 255 were generated as input symbols. One point to note is the number of symbols used for each test. Since the point of the test was to determine the frequency of accesses at each address, it is important to start with an input symbol set that is an integer multiple of the number of addresses. This at least provides the potential that each address will receive an equal number of accesses.

As previously discussed, several sets of input data were produced. Each of the resulting files was placed in its own directory. The C program histogra.c was then run on each of the these data files. Then, to ensure uniformity of analysis, a single Mathcad document was produced. This document was then copied to each directory and calculated. Table I shows the directories and how the data file in each directory was created.

Table I Input Data Sets

Directory (Appendix)	Random Values Produced				
	How Many	Range	Distribution	Seed Value	Method
CDEUNIF (B1)	4096	0-4095 12 bits	Uniform	5	Mathcad
SYMUNIF (B2)	4096	0-255 8 bits	Uniform	5	Mathcad
CDEGAUSS (B3)	4096	0-4095 12 bits	Gaussian	5	Mathcad
SYMGAUSS (B4)	4096	0-255 8 bits	Gaussian	5	Mathcad
CDEMYC1 (B5)	4096	0-4095 12 bits	Random	5	make-sym.c
SYMMYC1 (B6)	65536	0-255 8 bits	Random	5	make-sym.c

As an example, the data for SYMGAUSS will be discussed here. The complete documentation and analysis for each set of input values can be found in Appendix B, where each section of the appendix (as indicated in Table I) has the data for a different set.

Figure 1 shows part of the Mathcad document used to produce the gaussian distribution of random numbers that were used as symbols. To define this particular distribution there are four main values to select, the number of values to produce, and the

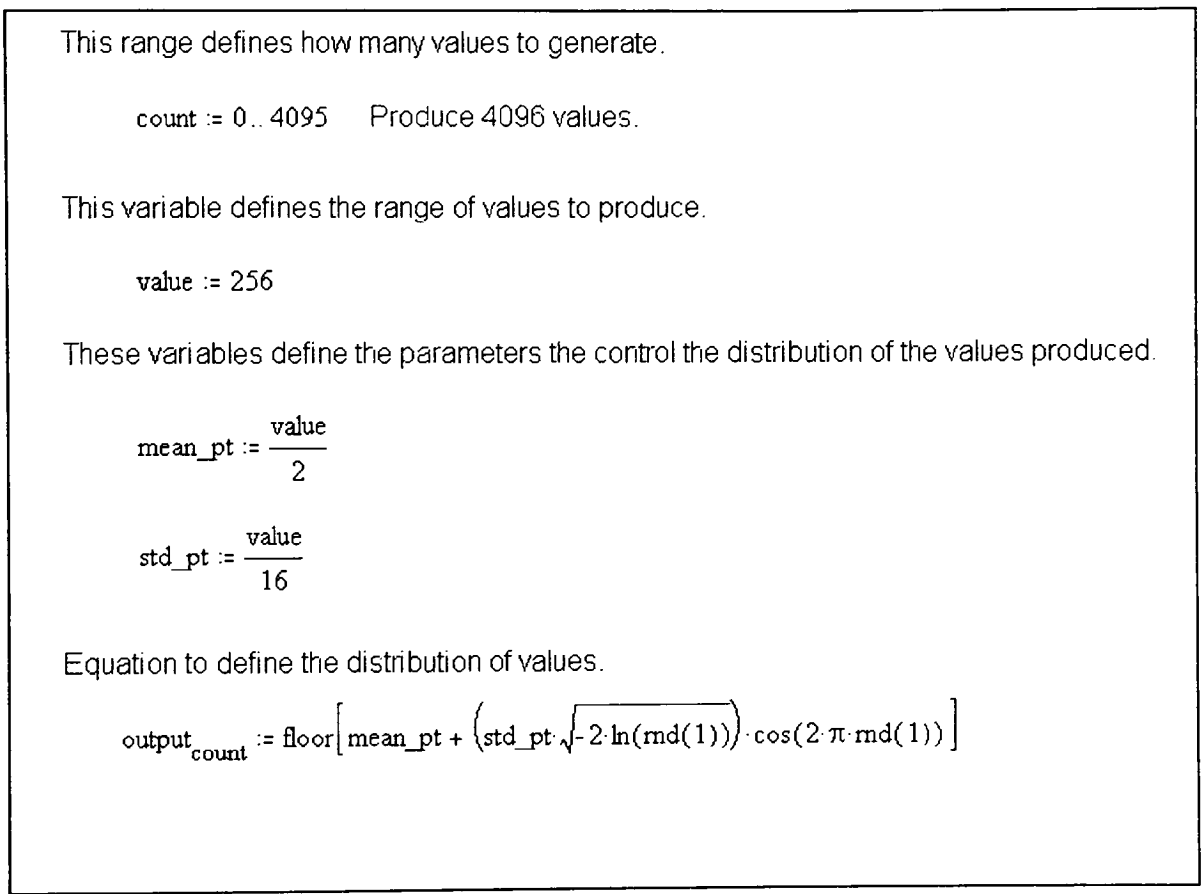


Figure 1 Equations for a gaussian distribution of random numbers

range in which the values will fall, and the mean and standard deviation for the of the distribution. These variables were determined by the algorithm parameters discussed above and testing. The equation shown is designed to produce a gaussian distribution. To produce the distribution, a mean and standard deviation had to be selected. The mean was picked as the center of the range (to make it easier to plot the results and verify that the distribution looked proper). The standard deviation was picked to be a small power of 2 (to make the distribution highly dissimilar to the uniform distribution). The actual equation used to

perform the calculation came from a Mathcad handbook (a Mathcad handbook is an electronic online document that is part of the tool). The next figure, Figure 2, generated from the Mathcad document shown in Appendix B4, shows the result of histogra.c processing the values produced by the document in Figure 1. As can be seen, and was expected, a gaussian distribution is indeed the result.

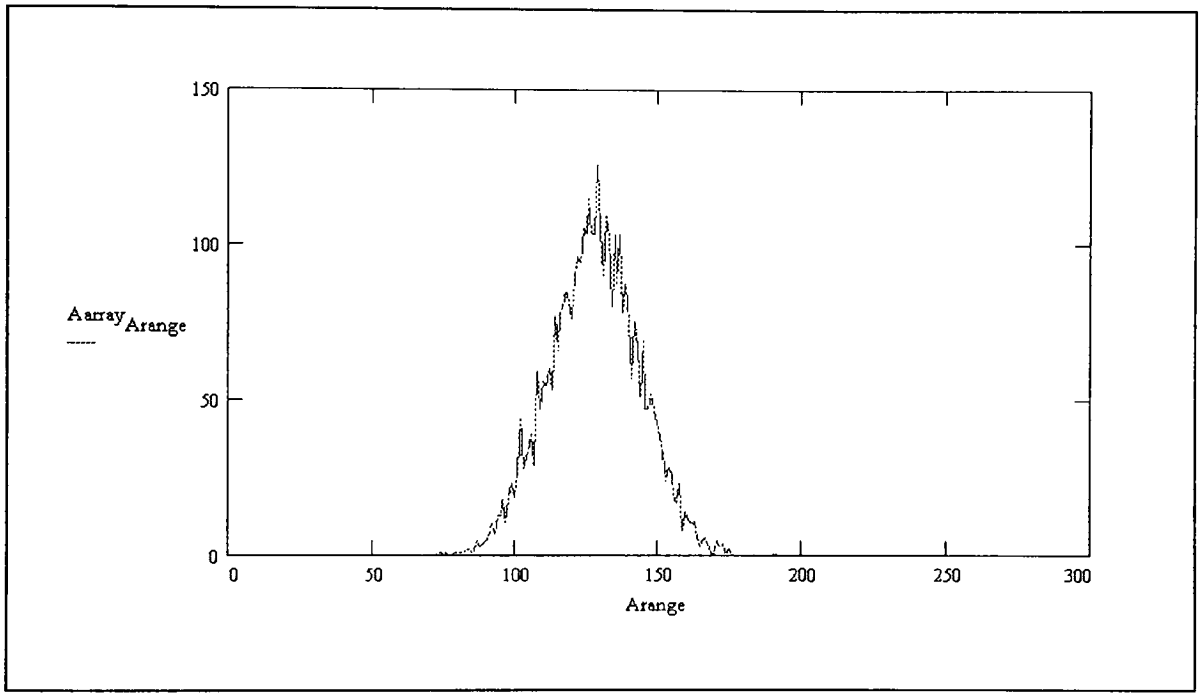


Figure 2 - Analysis of data produced by equations in Figure 1

3.3 DISCUSSION

Before the actual results are discussed, it is helpful to examine what results should be expected in the theoretical case. Two main types of results were produced by these tests. They were, the number of accesses at each memory location, and a histogram of how often

each location was accessed, which gave a frequency distribution. One benefit of using these outputs is the ease of illustrating the perfect, or theoretical, case.

For a perfect hashing function, every address would be accessed the same number of times. This would produce two very distinct graphs. For the accesses at each memory location, the theoretical graph is shown in Figure 3. As shown, this produces a flat horizontal line across the entire range of addresses, indicating equal accesses. The height of this line on the vertical axis is directly related to the integer constant times the number of locations used to determine how many symbols to produce. For the histogram, the theoretical graph is shown in Figure 4. This also produces a very distinctive graph, a vertical line. This shows that there was only one frequency of access, implying again that each location was

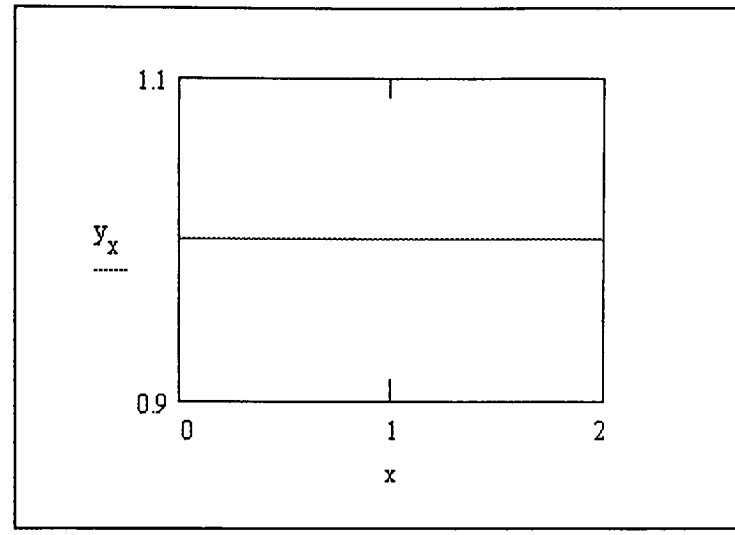


Figure 3 Theoretical plot of accesses at each location

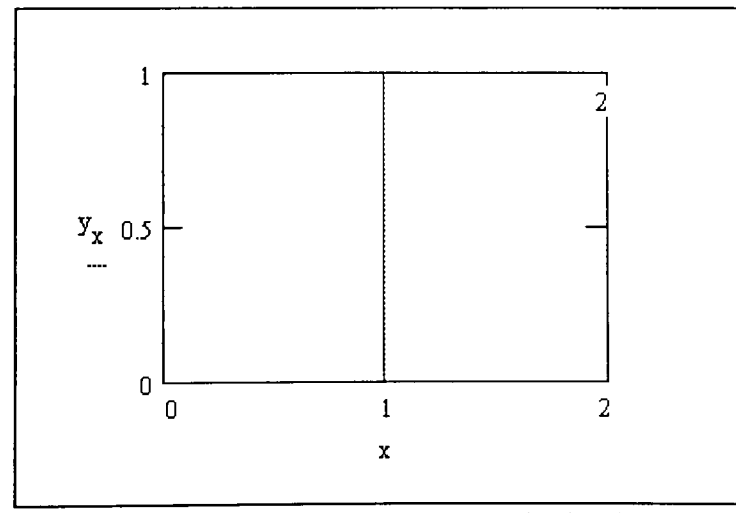


Figure 4 Theoretical plot of access distribution

accessed an equal number of times. The height of the line would be the number of locations, and the x coordinate would be the integer constant.

These two plots show the ideal case of a perfect hashing function, and the number of input symbols being a multiple of the number of locations. Any other set of conditions will cause both types of plots to degrade. It is this degradation that we are interested in because it can be used as a metric to compare hashing functions. It is not the amount of degradation that we are interested in, because there is no such thing as a perfect hashing function (even if the proper number of inputs were used), but rather the difference in degradation between the two selected hashing functions.

3.4 RESULTS

The input symbol and previous code files were tested against each other in all combinations. This was to observe the effect of a given type of symbol being used with similar and dissimilar types of previous codes. A method similar to that used when generating the input files was used. A separate directory was used for each test. The necessary symbol and previous code files were copied in from their creation directories. Then the addr-gen.c program was run in each directory. This produced a mod.prn and a xor.prn output files. Again, a Mathcad document was created, copied to each directory, and calculated, to ensure consistent data analysis. Table II shows the input types used, and the directory names in which the output was produced.

Table II - Output Data Sets (See Appendix)		Code File		
		CDEUNIF	CDEGAUSS	CDEMYC1
Symbol File	SYMUNIF	ADDRUCUS (C1)	ADDRGCUS (C2)	ADDRMCUS (C3)
	SYMGAUSS	ADDRUCGS (C4)	ADDRGCGS (C5)	ADDRMCGS (C6)
	SYMMYC1	ADDRUCMS (C7)	ADDRGCMS (C8)	ADDRMCMS (C9)

As with the input data only a small example of the results will be discussed here, the rest of the results will be in Appendix C. To keep the data sets reasonable, small amounts of data were used in most of the tests. While these results validate the point being demonstrated, that the two hashing functions perform similarly, the resulting plots are not as illustrative as they could be. The plots only show a one-sided distribution which is a direct result of the small size of the input symbol set. As larger symbol sets are used, the peak of the distribution moves away from zero and more fully illustrates the entire distribution, which would be more representative of the actual results expected in a real application. These results can be seen in some of the tests because one of the input files (SYMMYC1, Appendix B6) was generated with a larger number of symbols for this purpose.

A copy of figure C7-2 from Appendix C7 is shown in Figure 5. The plots in this figure, which show the results of analyzing ADDRUCMS, are included here for purposes of discussion. The upper plots show the number of accesses versus memory address, the upper of the two shows the results for the mod hashing algorithm, and the lower shows the same results for the XOR hashing algorithm. Due to the large number of data points, the plots are

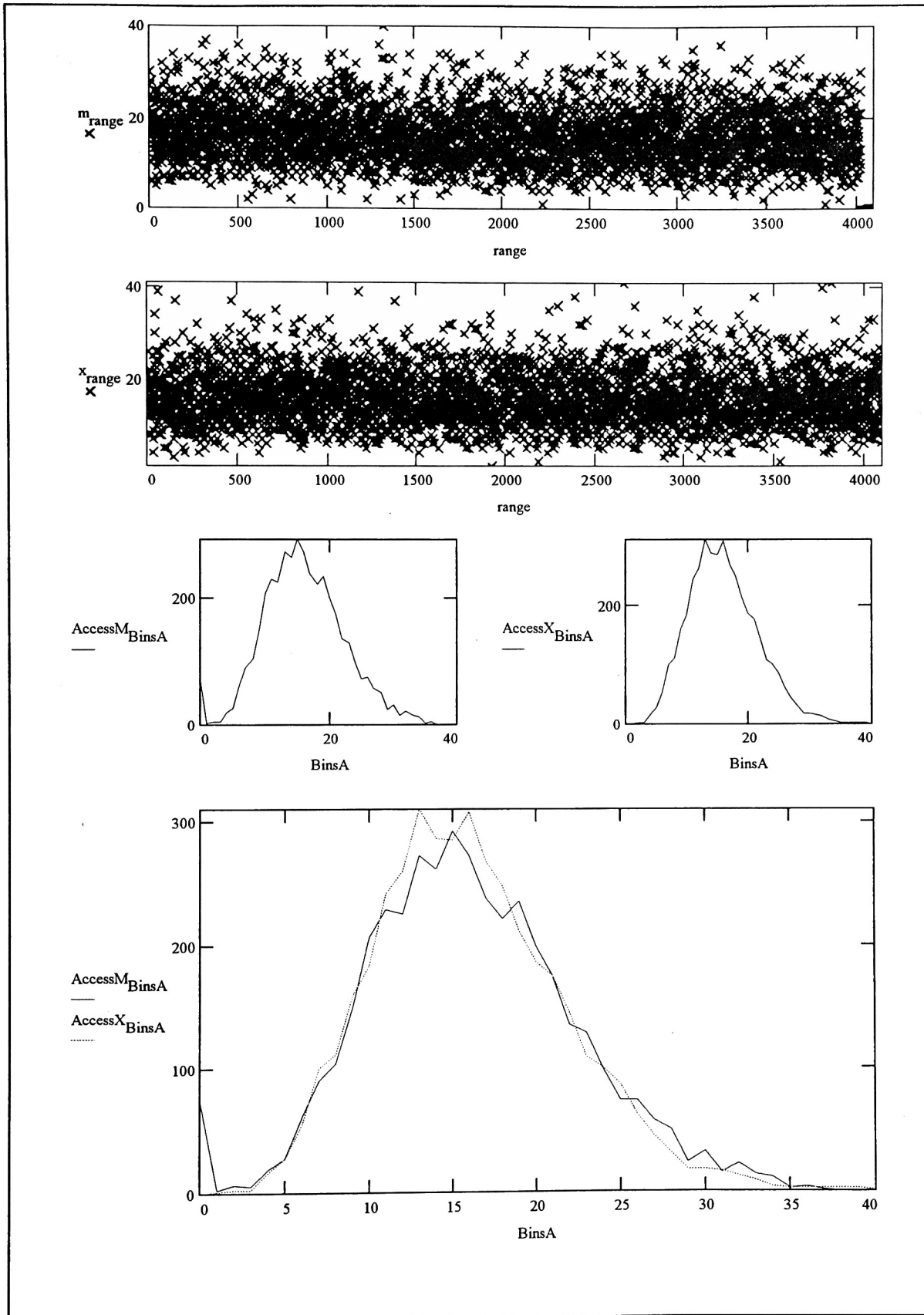


Figure 5 - Output data set ADDRUCMS (see Appendix C7)

very busy, but it can be seen that there is a definite horizontal band to the data centered on the mean of 16. To provide a better illustration of this, one set of data was processed using a convolution (smooth.c), and these results were processed by Mathcad to sum groups of data points. This reduced the clutter in the plot and gave a better view of the underlying structure. This extended analysis can be found in Appendix C9. The central two plots in the figure show the plots for the histograms of the hashing functions. The left most of the two plots shows the information for the mod hashing algorithm, and right most shows the similar information for the XOR hashing algorithm. For comparison purposes the last two plots were overlaid onto the same set of axis and this is shown in the bottom plot.

These plots show a definite gaussian distribution, which is a degradation of the theoretically expected vertical line. The plots show that neither of the hashing functions performs perfectly, which should be expected, but that they both demonstrate a degraded version of the theoretical plot. But, most importantly, both plots show that the algorithms function very similarly. This is what was being attempted, and the data shows that this result was obtained. This demonstrates that a new hashing function, which is easier to implement in hardware, has been developed and shown to work similarly to a commonly used, difficult to implement, software version.

There are several items to remember. First, even though it is known that hashing produces collisions, that effect has been ignored while developing the algorithms in this work. Second, the modulo hashing algorithm does not use all the available memory, while the XOR algorithm has the potential to do so.

Collisions have been ignored because they do not relate to the purpose of this study. Both algorithms will produce collisions, and as long as both distribute the addresses throughout the entire range in a similar fashion, the resulting functionality and collision frequency will also be similar. It should also be noted that not all collisions are undesirable. For example, if a string searching or matching algorithm is using the hashing function, the collision could show that a match has been found. This could be the desired result of the process. Therefore the number, frequency, and importance of collisions are more of a concern and function of the larger enclosing algorithm than the hashing algorithm.

The fact that the modulo does not use all of the available memory has already been mentioned. This seems to suggest that using a prime number as close to the table size as possible would be desirable. This turns out not to be the case. The prime chosen was selected from a table of recommended primes, and they are recommended for a good reason. This was verified by testing several other larger prime numbers (closer to the actual table size) in the algorithm, but they all produced more degradation (sometimes severe) from the theoretical. So, even though a smaller portion of the memory may used, if the accessed portion is used effectively, the overall performance is superior.

It should be remembered that these tests were not designed to measure the "goodness" or "badness" of the hashing algorithms. That type of testing is beyond the scope of this work. These tests were meant to check a specific set of constraints; does the XOR hashing algorithm perform similarly to the baseline modulo hashing algorithm. It would be dangerous to attempt to draw too many conclusions from the results, but I believe that this work shows

that the selected hashing algorithm performs well, and is very similar in results to the baseline algorithm.

4.0 VHDL IMPLEMENTATION

Before looking at the details of the particular hashing algorithm discussed in the previous sections, I will give a brief introduction and overview of VHDL. This will give some historical perspective to its application here, and provide the rational for its use in this and other designs. This will be especially significant with its regard to moving designs from the abstract world of theoretical algorithm development to the concrete world of hardware development.

VHDL is formalized in the IEEE specification 1076-1987 (the IEEE is currently working on the revised version, 1076-1992). VHDL is an acronym that itself contains acronyms. VHDL stands for; VHSIC Hardware Description Language. VHSIC stands for; Very High Speed IC. IC stands for; Integrated Circuit. The original impetus for VHSIC work came from the Department of Defense. The DOD was also one of the main forces behind the development of VHDL. Since that time, a much broader audience has developed for the use and standardization of VHDL.

The need for hardware description languages (HDLs) has long been recognized in industry, although many people will argue about what constitutes an HDL. Some refer to the early languages used to write simple state machines as the beginnings of HDL development, others claim that only the new languages are truly HDLs. The need for more powerful and compact methods of description became necessary as the complexity of designs began to increase, resulting in hand coding and simplification of the work becoming impossible. The designs would either take too long to process by hand, or the possibility of error became too large. This created the need to develop automated methods

of design and verification, implying the use of machines to do much of the work. Using machines meant that methods of describing and specifying the design in a machine "understandable" format had to be derived. This input also had to have the capability of being understandable to humans. Thus some of the early HDLs were born.

As time progressed and circuit complexity began rapidly increasing, it became obvious that the ability to describe larger pieces of designs, or even entire systems, would have great benefit. This drove the development of increasingly more capable HDLs. There were two main forces behind this. First, design complexity was increasing so rapidly that it was becoming difficult for designers to keep all the low-level details in mind, while at the same time the low-level details were becoming less necessary as technologies matured to the point of being virtually able to guarantee low level device functionality and correctness. Second, computers and the tools they fostered were becoming increasingly more sophisticated and capable, allowing them to automate larger and larger portions of the low-level implementation details freeing the designers to concentrate on the high-level system details.

Proprietary HDLs began to emerge that resulted in tying designers to a particular company's toolsets and methodologies. Verilog was one of the most popular HDLs in this class. The Department of Defense had major problems with this. It wanted to be able to have work developed at one company be transferable to another company without worrying about the tools and training of the companies and individuals involved. In addition, the DOD wanted to be able to have a closer correlation and verifiability between the specifications it distributed and the resulting products. This led to the development of VHDL which was to be an open standard, and was to be developed from the beginning as a language that could be used starting at the very earliest specification stages of a project, and carried through until the completion of the final design.

These desires led to the development of the language for DOD use, but the openness and the standardization quickly began to attract the attention of non-DOD industries. This drove the language development and standardization process into the IEEE, resulting in a standard that included a formal specification of VHDL. This is an important point that is often overlooked. VHDL is a completely formally specified language. All the constructs are defined and described in Backus Normal Form notation. This is very different from most other languages in common use. All the constructs are defined exactly; what they do, and how the relate to each other and the underlying language definition. For example, compare this to the popular programming language C. There is no formal specification for C (although there is some work in this direction). The language is described and some of its functionality illustrated in many books, the most famous being "The C Programming Language" by Kernighan and Ritchie. But this work does not give any formalism to the language. Anyone who wants to implement it is free to interpret the constructs and their interrelation differently. This is completely different than VHDL which uses the formalism of language constructs from programming language theory to describe it.

Looking at the preceding would seem to imply that all tools would be designed and perform identically. Unfortunately this is not the case. The specification was written by humans, and the development of tools from this language is done by humans. Being human means that there are bound to be errors, gray areas, interpretation differences and disputes, and other areas of confusion. All that the formal specification implies is that the differences should be minor and on the fringes of the specification rather than at the core of the language. This development has fostered broad support for VHDL among both vendors and users, both in the military and commercial sectors. This does not imply that

VHDL is without its problems or detractors, but its appeal continues to broaden and its use is rapidly raising with predictions that it will soon be the most common and most widely used HDL.

What makes VHDL more useful than many of the other HDLs? In addition to the standardization previously mentioned, VHDL is a higher level language. In fact, some of its detractors claim that it is too high level. They say that it has lost sight of the fact that the ultimate product is supposed to be physical devices. While it is true that there are some constructs which cannot be directly implemented in hardware, this begs the question. The more important question is what can VHDL be used for, and are the high level constructs necessary or helpful?

To answer this question, there are two ways of looking at the problem. First is the traditional or historical way. In this view, tools were developed and implemented to automate portions of the design tasks, especially the highly repetitive parts. In this perspective the criticism is probably correct. The high level constructs don't provide any additional help describing portions of the design to be implemented. But in the second view, which is attempting to break some of the historical bounds, the approach is different. It is no longer just important to describe the implementation of the design, it is also equally important to specify the design, and to test the design. These last two items can often make very good use of higher level constructs, and there is no need to worry about implementation. Constructs used to specify or test the design will most often remain in the abstract world of computers, programming, and software. This not only makes the high level constructs useful, it demands that they exist in order to provide the necessary tools for the job. Without higher level constructs, many essential activities would be extremely difficult if not impossible.

The second view of the problem is just beginning to be developed and understood. Very few (if any) people have made use of the full range of activities to completely implement a design in this new view. But using VHDL to do this process is precisely what it was developed for. The new view is attempting to tightly bind together all the steps of a design, from early specification on through testing the final implementation..

4.1 BEHAVIORAL IMPLEMENTATION

What is a behavioral model? What is a structural model? There is a fair amount of discussion, and controversy, about the definition of these two terms. A very restrictive definition of a behavioral model is one that does not create or use a clock, anything else is considered structural. A more liberal definition says that a behavioral model is one that does not instantiate any components. If the model instantiates components, then it is structural. To complicate matters, there are some people that try to further refine the definitions and draw more distinctions between the two models, even to the point of defining different models such as dataflow, etc. In this paper, to preempt the digression of this discussion into another version of "how many angels can dance on the head of a pin", I chose to use the latter definition above, namely that the distinction is drawn by the instantiation of components.

The first VHDL implementation of the algorithm, the behavioral model, was constructed without concern about implementing the design in hardware. This allowed modeling the algorithm in a manner which allowed comparison of not only the results, but also comparison to the functional steps in the C code implementation. In addition, it simplified constructing an environment to contain the algorithm for analysis and testing without getting buried in low level details. An added benefit is that a behavioral model

will most often execute much faster. This can be very advantageous in the early modeling stages where many simulations may need to be run.

Writing a model at the behavioral level should not be confused with writing a totally unconstrained model. While this is possible, it is not desirable or helpful. In the approach taken in this project, the "blue sky" type of thinking should have been accomplished while developing the algorithm and writing the C code. Once that is completed, a significant amount of information and knowledge about the algorithm and its workings is already in hand. This allows the behavioral model to be constructed so as to constrain the problem as much as possible. This can have several benefits. It can help to check the algorithm at the edges of its area of operation, and it can lead directly to knowledge of the hardware element sizes needed for the final implementation (e.g. bus widths, register sizes, etc.).

VHDL is very useful for doing the type of constraining mentioned in the previous paragraph. VHDL is a strongly typed language. This means that each object declared is given a type, and can only contain objects of that type. In addition to this, subtypes can be declared. This allows operations to be defined that will work on different subtypes as long as the base type is the same, and at the same time restricts the values that the object can contain. This type of restriction, by using subtypes, can be very valuable as a design tool. Any attempt to assign a value to a subtyped object that is not within its range will be an error. These can be either analyze time or run time errors (note: in the lexicon of VHDL, source code is analyzed, not compiled). The advantage this provides is if a desired object size is chosen, the simulations will prove if this size is adequate, or an error will result. The ability to find errors will be strictly limited by the thoroughness and ingenuity used in designing the tests.

Finding errors is the main purpose for simulating. This means that the strong typing of VHDL can be a big assist in finding design errors, especially some of the subtle errors involving lost bits from overflow conditions etc. Being able to accomplish this at the behavioral stage is even more useful because it is much easier to make corrections and changes. Finding the errors earlier may even change the design, or possibly call the design feasibility into question. It is interesting to note that traditionally, these types of errors were not found until simulating at the lowest structural level, which meant after all the schematics had been created. In the new approach, simulations with all the appropriate sizes and types can be done at the conceptual stage, long before schematics have even been thought about.

To test the algorithm being discussed in this paper it would be necessary to have a source of symbols, and some method to use the output (the addresses generated). In VHDL the design element that "contains" the element under test is called a testbench. This identified two of the main portions of the code to be written: the behavioral algorithm and the testbench. These two elements will contain most of the effort, and have the code that can be followed to see what the algorithm is doing. In addition to this, it was decided to separate some of the details to more clearly illustrate the essential points. In addition to making the main body of the code clearer, this is good coding practice. In VHDL, elements of the design that are abstracted are either other design elements, or subprograms, definitions, types, etc. This latter type of information is contained in a set of design units called a package.

The basic structures in VHDL are design units. There are five types of design units: the entity, the architecture, the configuration, the package declaration, and the package body. Package declarations and package bodies are very similar to subroutine modules in other high level languages. The main difference is that the declarations and

definitions are separated into different design units to allow individual analysis. Entities are the design units that represent "real" devices. That device may be a gate, a chip, a board, or a system. The function or size of the device is not important, what is important is the fact that it represents something physical that may or may not interface to the rest of the world. The entity is the object that contains the declaration of any interfaces into and out of the design unit (if any exist). An architecture is the definition of the functionality that will implement an entity. There may be any number of architectures for a given entity. If there are multiple architectures or other instantiated elements that need to be specified, the configuration is the design unit used. A configuration fully specifies what "component" to use for each element in the entity and architecture that it is configuring.

The testbench, called HashTB.vhdl, is shown in Appendix D1. For this work, it reads in the needed symbols from a file, it writes the addresses produced to another file, it produces a histogram of the addresses, it compares this histogram to a reference file that is read from disk, it stores the histogram back onto disk, and it produces the clock and enable signals needed to drive the component under test. It performs all these functions so that it is not only providing input to the algorithm, but it is also testing the results and comparing them to the original results from the C code. This shows some of the power of a testbench. Not only can it drive the simulation, but it can do much of the work that traditionally was performed by the designer looking at waveforms or output files. A skillfully constructed testbench can almost fully automate the simulation and checking cycle, relieving the designer of tedious and error prone tasks.

One of the main statements used in VHDL is the process. The process statement is used to encapsulate a section of code that will represent a specific action, or series of actions. Within each process the statements are executed sequentially. This sequential

execution is what defines the behavior of each process, giving it its identity or functionality. But sequential execution of a series of statements will not produce a realistic model of hardware where many things happen in parallel, simultaneously. Thus all processes (if more than one exist) execute concurrently. There are additional statements that can be used that will also execute concurrently with the processes. The importance of the elements at the concurrent level is the relationship between them, especially the communications between the various elements, because this is what creates the ability to realistically model hardware.

When developing the behavioral model, a number of decisions had to be made concerning the model. Of particular concern in this paper was the further development of the structural model and the desire to unify this work with the behavioral model in the most seamless fashion. This fact preordained the outcome of some of the decisions. One was the inclusion of a clock and enable signals in the testbench. Since it was desired to use the same testbench to drive both the behavioral and structural models, and the structural model was to be developed in a synchronous fashion, a clock was necessary. The enable signals were optional, but gave a more realistic design to the structural model.

Since the testbench is intended solely for exercising and testing of the instantiated component and is not going to be further developed or synthesized, this allowed a wide latitude to the developer. Many different code constructs are now available to the testbench developer that would be unusable when considering a design for real hardware, or synthesizability. In addition, special processes can be added that may have no purpose to the device under test, but may exist only to aid the developer or the tools being used. Some of these can be seen in the code for the testbench, shown in Appendix D1. There are concurrent statements to produce a clock, and initiate operations. There is one process

that exists only to terminate the clock when the simulation has completed, which exists to aid the simulator. The two processes that do most of the work are ReadFile, and WriteFile. The names are self explanatory, with the addition that WriteFile also generates a histogram of the addresses produced. The last process, CompareArray, takes the histogram and compares it against the histogram produced by the C code, and stores the VHDL produced histogram back to disk. This final process exists to simplify the simulation and comparison process. It allows the simulations to be run in a batch mode from the command line, with the testbench doing all of the work. Of course, this will only work after the testbench and component have been debugged.

To keep the testbench, and the algorithm design units from getting too cluttered, some of the supporting types, declarations, subprograms, and component instantiations were put in a package. The code for this package, called HashPKG.vhdl is shown in Appendix D3. The package contains both the package declaration and the package body. While this is not necessary, since each is a separately analyzable design unit, it was done here since the package is not that large. As can be seen, the package contains a variety of things that can be used by several of the different design units. In addition to this, if a package is well written and documented, it may often be useful for more than one design, proving one of VHDL's design goals, code (design) reuse. In the case of HashPKG.vhdl the code was written only with this project in mind, but the point is illustrated by another package called dataconv_pkg.vhdl, shown in Appendix D4, that is used by HashPKG but was not written for this project. dataconv_pkg was a package that was written for previous work, but used as it was for this work. In this fashion, as designers gain experience with VHDL, they may build up a library of packages and design units that will allow future work to be accomplished much more rapidly because parts of it can be put together from existing components.

The discussion has now touched on all the major design units used in the behavioral design, except for the algorithm itself. The discussion progressed in this fashion, because it approximates the manner and order in which the design units were actually constructed. Often it is best to build the tools and support infrastructure before tackling the main design unit. This allows full effort to be applied to the element being designed, without the effort being diluted by other concerns. In this case, the testbench was developed first, until it could read, write, and compare test files. This caused much of the package to be developed in parallel. By the time the algorithm development began, most of the supporting code existed. While it will usually be impossible to fully develop all the supporting code before starting on the main element, small changes and additions will be trivial matters if most of the supporting code and the hierarchy exists. The full hierarchy of the behavioral model can be seen in Figure 6.

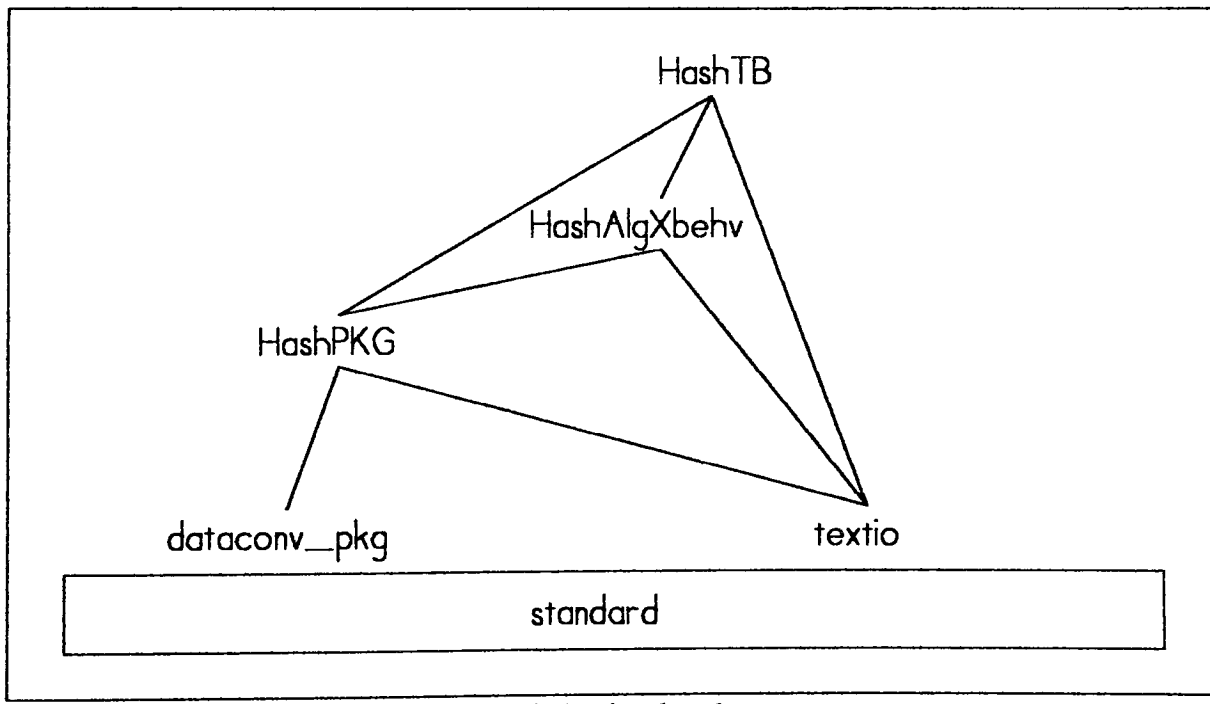


Figure 6 - Hierarchy of objects for the behavioral code

When the testbench was working and could do all the necessary file manipulations, work was started on HashAlgXbehv.vhdl, shown in Appendix D6. This design unit contains the code for the behavioral version of the XOR hashing algorithm. As can be seen, it is relatively compact and straightforward, consisting of four processes. This shows that proper use of testbenches, and abstracting support functions to packages, can make the main design unit much simpler, allowing easier development, debugging, and allowing it to be more self-explanatory.

Of the four processes, only one is concerned with the actual algorithm. The other three are support functions. The NextCode process is needed to prime the system. The very first symbol read in becomes the first previous code, after that the previous code always comes from the memory. HashMem is a process the simulates a ROM type of device. Whenever an address is presented, the process returns the proper data for that address. This process also has some ancillary work to do; it has to know the first time that it runs (elaboration time), and fill the array from a file. Enable is a process that watches for valid data to be presented, and after the proper delay, signals that valid data is available on the output. The last process is the one concerned with the actual hashing operation. When seen at the behavioral level, the processing requires a very minimal amount of code, which shows the power of high level programming. Each line contains a number of high level operations, functions, and/or array operations.

When all the code development was complete, the behavioral algorithm was tested via the testbench. A Unix script was written that ran the algorithm written in C code. The output from this was stored in a file. The script then ran the testbench which ran the VHDL version of the algorithm. The testbench also read the file that had been produced by the C code. It used this as the basis for comparing the values produced by the VHDL.

The values produced by the two methods had to compare exactly. These tests were run on all the possible combinations of input codes and input addresses that had been generated.

4.2 STRUCTURAL IMPLEMENTATION

When doing the structural implementation, it was desired to use the testbench and packages that had been developed for the behavioral version without any modification other than the additions of new components or subprogram declarations that may be needed. The second goal was to implement the algorithm in a completely structural manner, meaning only component instantiations and the interconnections between them. This second goal was bent slightly to show some of the tools of VHDL, port conversions and signal name changes.

Using VHDL, it is possible to have a single entity and write multiple architectures for it. That would have been possible, but was not done in this case. There were several reasons for this. First, it was desired to use integer base types as much as possible in the behavioral version, including many of the ports. Second, since it was desired to make the structural version only instantiations and interconnections, it was necessary to have ports that were based on bit or bit_vector types. To meet these goals, and the goal in the previous paragraph of using the testbench without modification, a new level of hierarchy had to be introduced. The testbench instantiates a component with the same ports as the behavioral version, but this is a "dummy" component that only does type conversions, and then instantiates the structural component. This intermediate component is called HashConvert, and its code is called HashConvert.vhdl which is shown in Appendix D5. The code for the structural version of the XOR hashing algorithm is called

HashAlgXstru.vhdl and is shown in Appendix D6. The full hierarchy of the structural model can be seen in Figure 7.

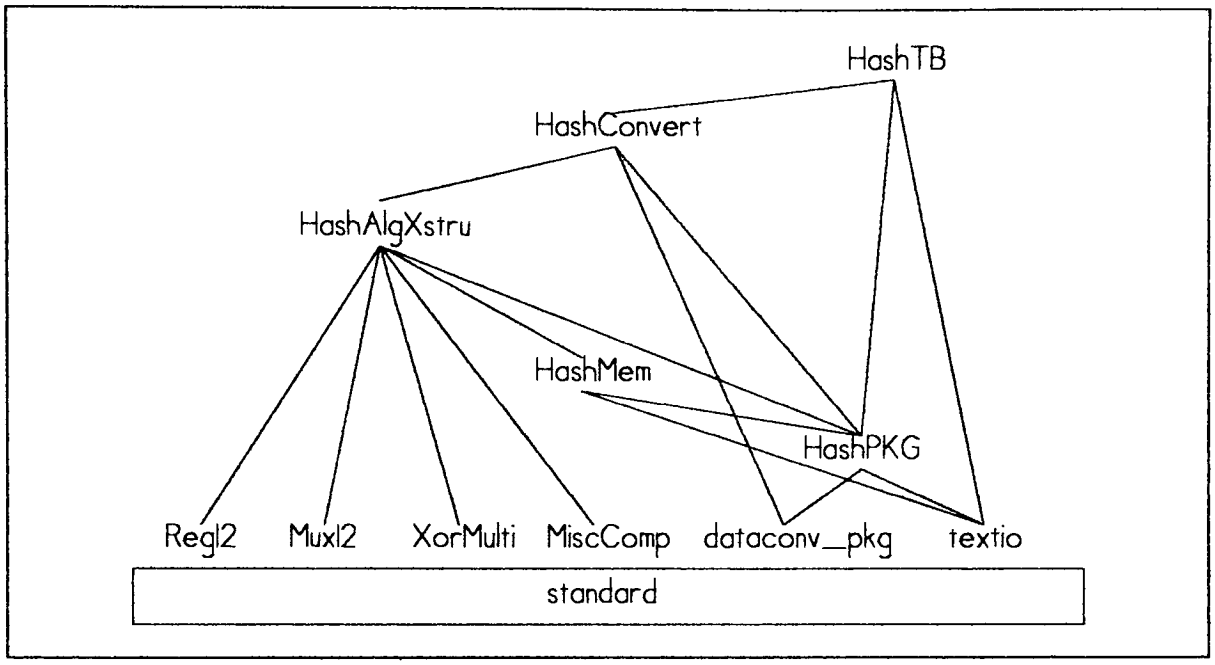


Figure 7 - Hierarchy of objects for the structural code

When looking at the structural code, the first reaction is, 'why?'. It is about as informative as trying to read a netlist; possible, but time consuming, error prone, and pointless. It merely lists the components used, and the connections to their ports. It is much better to have the structural code written by a machine, it is faster and there are fewer errors. The programs used to do this are called synthesis tools. In the code there are two places where signals are assigned to other signals, identifier changes needed to satisfy VHDL's port mode requirements. The only real point of interest is on the component HashMem where port conversion functions are used.

The use of the port conversion functions was not essential, but it demonstrates one of VHDL's features and allowed code reuse. The memory element needs to read a file of values that represent the previous codes. These values were produced using the C program previously mentioned. Since it is not possible to directly produce a bit_vector from a C program and it is possible to produce ASCII values, that is how all the output was stored in files. This also allowed for greater tool and platform independence, as ASCII is standard across almost all platforms while binary file formats are frequently different. Since the data stored was ASCII, and VHDL only allows indexing arrays with integers or enumerated types, the memory element was constructed as a high level element. The behavioral version of the algorithm implemented the file open and memory reading directly. In the structural version this was undesirable, since there is no piece of hardware that corresponds to opening a file. To reuse the code from the behavioral version, a memory component was constructed with ports that were a subtypes of integer. Then to connect this component to the bit_vectors used in the structural code, port conversion functions were needed. These are special functions that accept a signal in the port map, and return a signal of a different type. This allowed conversion to and from bit_vectors in the structural algorithm and the integers required by the memory element. This would not have been necessary, the conversion could have been done inside the memory element, but this method allowed direct reuse of the code from the behavioral version of the algorithm.

A structural description of a device does not provide much information beyond the components used and the interconnection. The real information on functionality lies at a lower level. There could be a hierarchy of structural elements, or, as in this case, the next lower level contains all the functionality. In this example, the devices at the lower level were implemented as functional components: the hash memory (already discussed), a 12 bit register called Reg12.vhdl which is shown in Appendix D8, a 12 bit multiplexor called

Max12.vhdl which is shown in Appendix D9, a variable width XOR structure called XorMulti.vhdl which is shown in Appendix D10 (more about this later), and several other components representing AND gates, inverters, and a single bit register which are in a package called MiscComp.vhdl and is all shown in Appendix D11.

These components were chosen as the best fit for this algorithm. Other components, or other size components, could have been used. A decision on this could depend on the desired implementation of the design being described. If the design is going to be built as a printed circuit board using standard parts, then the parts chosen should represent the parts actually available. If the design is going to be in programmable devices, or an ASIC, then it is usually possible to build them to the desired custom size.

One component deserves more explanation. That is the one labeled XorMulti.vhdl, that was referenced earlier as the variable width XOR structure. This component shows another feature, and some of the power of VHDL. It lets the tools do the work and frees the designer from some of the details. This device uses the VHDL generate statement. This statement will repeat a structure until it is the desired size. In this case, several different size XOR comparisons were needed. To achieve this, unconstrained arrays were used as the ports into the component. At instantiation time, the size of the arrays will become known by virtue of the size of the connecting group of signals. This information is fed to the generate statement which allows it to build a component of the exact size needed. Looking at the structural algorithm description, it can be seen that at the higher level no special consideration is needed. The component is simply interconnected the same as any other, with the exception that the number of inputs and outputs are not restricted to a fixed size.

The structural implementation illustrates another point about how this algorithm is tailored for hardware implementation. In the past, designers were mainly concerned with the signal delay through the active circuit elements. As more designs are moved into LSI devices, and at the same time the delay through each LSI element is decreasing, the active element delay is becoming less of a concern and the interconnect delay is becoming more of a concern. In many of the latest generation small geometry devices, the interconnect delay is becoming larger than the active element delay. Interconnect delay cannot be designed out of a device, the circuit elements have to be connected together. The algorithm being developed here uses "bit shuffling" as part of its strategy. This is a function that is implementable using the interconnect. As the saying goes "if you can't beat'em, join'em", which in this case means that even though the interconnect delay can't be designed out, the interconnect can be made part of the algorithm at no additional circuit or delay cost. This uses all available resources to the fullest, even using the interconnect for functionality above and beyond its normal prosaic duty of active element connection.

4.3 IMPLEMENTATION NOTES

There are a number of additional notes about the implementation of the various pieces of code. One of the goals of this work was to produce an exact duplication of results, not just between the C code and the VHDL, but also when the algorithms were executed on different platforms. This goal is actually much more easily obtained using VHDL. This is due to the strong typing, especially when exact size subtypes are used. The fact the language allows the user to specify what each type will contain, and that the language restricts each implementation to exactly adhere to this, eases platform dependency

concerns. Some care will need to be taken if only the native unconstrained types are used, as these can have different limits on different machines.

Unfortunately C is not strongly typed. The types in the C language are not defined, they are all dependent on the particular implementation on each platform. As an example, one of the common differences is that the type *int* can be 16 or 32 bits. This can be of concern for any code that performs low level operations, such as the algorithm described here. In the shifting operations used, care needed to be taken to ensure that the size of the type being operated upon was large enough to hold the intermediate value generated. In addition to this, there is also the concern about how the memory is structured. The competing memory layouts are typically referred to as big-endian and little-endian. This concern becomes important when doing bit operations. It is impossible to perform the operation correctly unless it is known where the actual bits being operated upon are stored within the particular variable. This had to be dealt with in the procedure that did the bit reversing calculations in the C code.

5.0 CONCLUSION

The goal of this work was to develop a key generation algorithm that would be efficient to implement in hardware, to verify its efficacy, and to then implement this algorithm using VHDL in both a behavioral and a structural style. The particular class of key generation algorithms considered were those commonly referred to as hashing algorithms. To verify the efficacy, a particular hashing algorithm that was representative of the type commonly used in software implementations was chosen for comparison.

The first step in the work was the selection of the algorithm for comparison. From the literature, and examples of programs examined, a modulo based algorithm using a prime modulus was chosen. After this, by experimentation and knowledge of the desired results, an algorithm was designed that would be more efficient to implement in hardware. To accomplish this, simple bit manipulations and XOR operations were used. A study was then undertaken to demonstrate the XOR algorithm was able to produce distributions of keys that were, at worst, not significantly less well distributed than the baseline modulo algorithm. When this had been finished, VHDL was used to design a test environment, and an implementation of the algorithm, at a behavioral level. The test environment was used to verify that the VHDL algorithm performed *exactly* the same as the version in C code. To complete the design work, the portion of the design that represented the algorithm was then redesigned to a structural VHDL level. Again this was checked using the same test environment to verify that it performed ***exactly*** the same as the baseline.

The result of this work is a algorithm that can be implemented in hardware using very

simple structures, indeed even the interconnect between the structures as part of the algorithm. This algorithm has been tested against data designed to exersice its capabilities across a range of inputs, and shown to perform as well as, and in selected cases, possibly better than the equivalent version implemented using software constructs.

A possible flow for taking an algorithm from the abstract design world to the concrete implementation world is illustrated by the methodology. While this is not the only method, and possibly not even the best, it illustrates the various stages that have to be accomplished. The final result, VHDL, is a solution that is becoming increasingly popular. There are a number of reasons for this. It is standardized, it can be executed and verified early in the design cycle before an implementation is chosen, and it can offer direct comparison to implementations in other languages including operating from the same data files.

Some of the usefulness and versatility of VHDL can be seen in the work done here. The ability of the language to describe functionality at different levels is illustrated, as is its power to create components that represent physical hardware. It is also obvious that another conclusion can be drawn about structural VHDL. **Don't**. It takes a lot of effort to write and is difficult to understand. The solution to this is to use synthesis tools, but a discussion of them is beyond the scope of this work.

APPENDIX A - C Source Code Files

Appendix A1 - Source Code for make-sym.c

This appendix contains the source code listing for the C language file make-sym.c. The purpose of this program was to generate pseudo random numbers to be used as symbols and previous codes. These values were written to output files to allow further processing by other programs.

The program operated by using the standard C language function rand(). For each value the program called the rand function in twice. The first value determined the number of times to call it to get the value to return. The analysis of the data produced in this program is shown in Appendices B5 and B6.

PROGRAM PARAMETERS

program name	
first argument	how many output values to produce
second argument	name of the output file
third argument	seed value for the random number generator
fourth argument	switch used to determine if 8 bit or 12 bit values should be produced

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*                                                                        */
/*                                                                        */
/*                  Copyright (C) 1992 by James Goeke                     */
/*                        All rights reserved.                            */
/*                                                                        */
/*                                                                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */


#define HOW_MANY_ARGS 5     /* number of command line arguments expected */
#define PROGRAM_NAME 0      /* position of program name argument */
#define VALUE_ARG 1         /* position of argument for number of values to create */
#define OUT_FILE_ARG 2      /* position of output file name argument */
#define OUT_W_FILE 0        /* position in array of output file name */
#define SEED_VALUE_ARG 3    /* position of arguement for random number seed */
#define SIZE_SWITCH 4       /* position of switch to determine the bit size of the values to create
*/
#define HOW_MANY_FILES 1    /* number of files program opens */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char *argv[]);

void main(int argc, char *argv[])
{

    int     f_cnt   0,          /* how many files are open */
            size,
            clos_tst,           /* error checking */
            clos_err = 0;       /* error checking */

    long    innumber;
    char    *endptr;

    FILE    **fname;                        /* array of file names */
    void    my_rand(long int *name);        /* function */
    void    usage( char *pgrm_name[] );     /* function */

    unsigned long int   tot_out_count = 0,
                        out_count;

    unsigned int    sym8_8(void),           /* function */
                    sym12_12(void),         /* function */
                    mlp,
                    (*fctn_ary[2])(void),   /* array of pointers to functions */
                    value = 0;              /* value being generated */

/*********************************************************************/


        /* very basic command line check, mainly to print usage if no args given */
    if( argc < HOW_MANY_ARGS ){
        usage( argv );
        exit(3);
        }

        /* assign the functions to their positions in the array to allow selection */
    fctn_ary[0] = sym8_8;
    fctn_ary[1] = sym12_12;

        /* check to find out if symbols or previous codes are to be generated */
    if ( (*argv[SIZE_SWITCH] == 's') || (*argv[SIZE_SWITCH] == 'S') ){
        size = 0;
        }
        else if ( (*argv[SIZE_SWITCH] == 'c') || (*argv[SIZE_SWITCH] == 'C') ){
        size = 1;
        }
        else{
            perror( "\nCOMMAND LINE ERROR" );
            fprintf( stderr, "\n\tINVALID COMMAND LINE SWITCH USED\n" );
```

```
        usage(argv);
        exit(10);
        }

    /* open array to store pointers to all the needed files */
fname   ( FILE ** ) calloc( (HOW_MANY_FILES), (unsigned int) sizeof( FILE * ) );
if( fname == NULL ){
    perror( "\nMEMORY ALLOCATION ERROR" );
    fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR FILES\n" );
    exit(4);
    }

    /* open output file */
if( (fname[OUT_W_FILE]  : fopen(argv[OUT_FILE_ARG], "W")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING OUTPUT FILE:  %s\n", argv[OUT_FILE_ARG]);
    perror( strcat("\nOUTPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(1);
    }
f_cnt += 1;

    /* read in how many values to generate */
    /* will have to use strtod on UNIX systems */
innumber = strtol(argv[VALUE_ARG], &endptr, 10);

    /* check for error in converting 'values to generate' argument */
    /* NULL means no conversion error */
if (*endptr == NULL){
    fprintf( stderr, "\n%ld  Values will be produced.\n", innumber);
    }
else{
    fprintf( stderr, "\n"
    "Error occurred converting input count to number.\n"
    "Character that stopped conversion: %c\n", *endptr);
    exit(7);
    }

    /* read in seed value for the random number generator, and use it */
    /* note:  no error checking here, atoi will return 0 on error */
srand( (unsigned) (atoi(argv[SEED_VALUE_ARG])) );

    /* operator error checking, make sure program is using what you think */
fprintf(stdout, "\nSeed value being used is:  %d.", (atoi(argv[SEEO_VALUE_ARG])) );

fprintf( stderr, "\n\n\tProgram is running.\n\n" );


    /* this loop does all the actual work of generating values */
do  {
    value = (*fctn_ary[size])();
        /* need to delete \n from last value to prevent downstream errors */
    out_count = fprintf (fname[OUT_W_FILE], "%d\n", value);
    if( out_count == EOF ){
        perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_W_FILE] );
        clos_err   2;
        }
    tot_out_count += 1;
    }
    while( tot_out_count < (innumber   1) );

    /* do last value without any line feed/carriage return in output file */
value = (*fctn_ary[size])();
out_count = fprintf (fname[OUT_W_FILE], "%d", value);
if( out_count == EOF ){
    perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_W_FILE] );
    clos_err   2;
    }
tot_out_count += 1;


fprintf( stderr, "\nTotal values written out:  %lu\n", tot_out_count );
```

```
        /* close all the open files */
    for( mlp = 0; mlp < f_cnt; mlp++ ){
        clos_tst   fclose( fname[mlp] );
        if( clos_tst == EOF ){
            perror( strcat("FILE CLOSE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
            fprintf( stderr, "\n\tERROR CLOSING FILE:  %s\n", fname[mlp] );
            clos_err   1;
            }
        }

        /* free all allocated memory */
    free( fname );

    if( clos_err == 1 )
        exit(2);
    else if(clos_err == 2)
        exit(5);
    else
        exit(0);

}


/*****************************************************************/


    /* message to print if wrong number of arguments were given */
void usage( char *fargv[] )
{

fprintf( stderr, "\n\n"

"This program generates pseudo random 8 bit values to be used as 'symbols',\n"
"or 12 bit values to be used 'previous codes' (selected by command line switch).\n"
"Please choose S for 8 bit values, and C for 12 bit values.\n\t"
    "These values are written to an output file.\n\n "

"THE CORRECT INVOCATION SYNTAX IS:\n\
%s\n\thow_many_values_to_create\n\t\toutput_file_name\n\t\t\t\
rand_seed_value\n\t\t\t\tvalue_size_switch\n\n", fargv[0] );

}


/*****************************************************************/


    /* function to try to randomize the rand function */
void my_rand(long int *new_rand)
{
    int rand(void),     /* the random function from library */
        lpr;            /* loop variable */

    long int num;       /* how many times to call rand to get a random number */

    num = (rand() % 50);

    for(lpr = 0; lpr <= num; lpr++){
        *new_rand = rand();
        }

}


/*****************************************************************/


    /* function to build 8 bit values from 2 pseudo random numbers */
unsigned int sym8_8(void)
{

    int sym1 = 0,
        sym2   0;
```

```
    long int ran_num;

    (void) my_rand(&ran_num);
    sym1 = (int)((ran_num >> 5) % 16);
    (void) my_rand(&ran_num);
    sym2 = (int)((ran_num >> 5) % 16);

    return ((sym1<<4) + sym2);
}


/*********************************************************************/


    /* function to build 12 bit values from 2 pseudo random numbers */
unsigned int sym12_12(void)
{

    int sym1 = 0,
        sym2 = 0;

    long int ran_num;

    (void) my_rand(&ran_num);
    sym1 = (int)((ran_num >> 5) % 64);
    (void) my_rand(&ran_num);
    sym2 = (int)((ran_num >> 5) % 64);

    return ((sym1 << 6) + sym2);
}


/*********************************************************************/
```

Appendix A2 - Source Code for histogra.c

This appendix contains the source code listing for the C language file histogra.c. The purpose of this program was to generate histograms of how many times each one of the input values was produced. This frequency data was written to an output file to allow further processing by other programs.

The program operated by opening an array of the appropriate size (determined by a command line switch). It read each input value and incremented the count at the array location for that input value.

PROGRAM PARAMETERS

program name	
first argument	name of the input file
second argument	name of the output file
third argument	command line switch used to determine the size of the array to use

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*                                                                        */
/*                                                                        */
/*                  Copyright (C) 1992 by James Goeke                     */
/*                         All rights reserved.                           */
/*                                                                        */
/*                                                                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */


#define HOW_MANY_ARGS 4     /* how many arguments expected on command line */
#define PROGRAM_NAME 0      /* argument number of program name */
#define IN_FILE_NAME 0      /* position in array of pointer to input file */
#define IN_FILE_ARG 1       /* argument number of input file name */
#define OUT_FILE_NAME 1     /* position in array of pointer to output file */
#define OUT_FILE_ARG 2      /* argument number of output file name */
#define SIZE_SWITCH 3       /* argument number of character indicating what size values to histogram
*/
#define HOW_MANY_FILES 2    /* how many files the program opens */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]);

void main(int argc, char *argv[])
{
    FILE    **fname;        /* array of pointers to files */

    int     f_cnt   0,
            mlp,
            lp,
            size,
            out_count   0,
            clos_tst,
            clos_err   0;

    long int    *hist;

    char    msg[10];
    void    usage( char *pgrm_name[] ); /* function */

/*********************************************************************/


        /* very basic command line check, mainly to print usage if no args given */
    if( argc < HOW_MANY_ARGS ){
        usage( argv );
        exit(3);
        }

        /* check to find out if symbols or previous codes are to be histogramed */
    if ( (*argv[SIZE_SWITCH] == 's') || (*argv[SIZE_SWITCH] == 'S') ){
        size = 255;
        }
        else if ( (*argv[SIZE_SWITCH] == 'c') || (*argv[SIZE_SWITCH] == 'C') ){
        size   4095;
        }
        else{
            perror( "\nCOMMAND LINE ERROR" );
            fprintf( stderr, "\n\tINVALID COMMAND LINE SWITCH USED\n" );
            usage(argv);
            exit(10);
            }

        /* open array for the histogram */
    hist = (long int *) calloc( (size + 1), (unsigned int) sizeof( long int ) );
    if( hist == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR HISTOGRAM\n" );
        exit(7);
        }
```

```
    /* open array to store pointers to all the needed files */
fname = ( FILE ** ) calloc( (HOW_MANY_FILES), (unsigned int) sizeof( FILE * ) );
if( fname == NULL ){
    perror( "\nMEMORY ALLOCATION ERROR" );
    fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR FILES\n" );
    exit(4);
    }

    /* open input file */
if( (fname[IN_FILE_NAME]   fopen(argv[IN_FILE_ARG], "r")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING INPUT FILE:  %s\n", argv[IN_FILE_ARG]);
    perror( strcat("\nINPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(1);
    }
f_cnt += 1;

/* display the input file name to the user */
fprintf(stdout, "\n Input file is:\t\t%s\n", argv[IN_FILE_ARG]);

    /* open output file */
if( (fname[OUT_FILE_NAME]   fopen(argv[OUT_FILE_ARG], "w")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING OUTPUT FILE:  %s\n", argv[OUT_FILE_ARG]);
    perror( strcat("\nOUTPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(3);
    }
f_cnt += 1;

/* display the output file name to the user */
printf("\n Output file is:\t%s\n\n", argv[OUT_FILE_ARG]);

/* display message to acknowledge that something is happening */
fprintf( stderr, "\nProcessing the input file.\n");
fflush( stderr );

    /* read first string from the input file */
if (NULL == fgets(msg, 10, fname[IN_FILE_NAME]) ){
    printf("\n\t ERROR:  no data in file!\n\n");
    exit(1);
    }
else{
    /* this section is where all the work is accomplished */
    do {
        /* the modulo is to prevent lots of error checking on the input values */
        hist[atoi(msg) % (size + 1)] += 1;
        }
        while (NULL != fgets(msg, 10, fname[IN_FILE_NAME]) );
    }

for (lp = 0; lp <= (size   1); lp++){
    /* need to delete \n from last symbol to prevent downstream errors */
    out_count   fprintf (fname[OUT_FILE_NAME], "%d\n", hist[lp]);
    if( out_count == EOF ){
        perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_FILE_NAME] );
        clos_err = 2;
        }
    }

/* do last output without any carriage return/line feed */
/* code was put inline for speed (could have used an if clause above) */
out_count = fprintf (fname[OUT_FILE_NAME], "%d", hist[size]);
if( out_count == EOF ){
    perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_FILE_NAME] );
    clos_err   2;
    }

for( mlp   0; mlp < f_cnt; mlp++ ){
    clos_tst   fclose(fname[mlp]);
        /* EOF is -1 */
    if( clos_tst == EOF ){
        perror( strcat("FILE CLOSE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR CLOSING FILE:  %s\n", fname[mlp] );
        clos_err   1;
```

```
            }
        }

        /* free all allocated memory */
    free( hist );
    free( fname );

    /* display termination message to the user */
    fprintf(stdout, "\n Successfully finished processing:\t%s\n", argv[IN_FILE_ARG]);

    if( clos_err == 1 )
        exit(2);
    else if(clos_err == 2)
        exit(5);
    else
        exit(0);


}

/*********************************************************************/


    /* message to print if wrong number of arguments were given */
void usage( char *fargv[] )
{

fprintf( stderr, "\n\n "

"This program histograms data from an input file, \n\t "
    "and writes the data to another output file.\n\n "

    "The program works with 8 bit or 12 bit symbols,\n"
    "\tso 256 bins or 4096 bins are used.\n\n "
"Please choose S for 8 bit values, and C for 12 bit values.\n\n"

"BE CAREFUL, ANYTHING OVER 12 BITS WILL CAUSE ALIASING IN THE DATA!\n"
"\t(or choosing the 8 bit switch for 12 bit data)\n\n"

"THE CORRECT INVOCATION SYNTAX IS:\n \
%s input_file_name output_file_name size_switch\n\n", fargv[0] );

}

/*********************************************************************/
```

Appendix A3 - Source Code for smooth.c

This appendix contains the source code listing for the C language file smooth.c. The purpose of this program was to perform a convolution on the input data. This convolved data was written to an output file to allow further processing by other programs.

The program operated by opening an array of the appropriate size (determined by a command line switch). It only worked with 12 bit data values, but the array size was also determined by the size of the convolution window to be used. It required an odd value for the convolution window size to assure that there would be a central value that the convolution result would be placed into. The convolution was an average of all the values in the window.

PROGRAM PARAMETERS

program name	
first argument	name of the input file
second argument	name of the output file
third argument	size of the window to be used for the convolution (only odd values accepted)

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*                                                                        */
/*                                                                        */
/*                  Copyright (C) 1992 by James Goeke                     */
/*                         All rights reserved.                           */
/*                                                                        */
/*                                                                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */


#define HOW_MANY_ARGS 4      /* how many arguments expected on command line */
#define PROGRAM_NAME 0       /* argument number of program name */
#define IN_FILE_NAME 0       /* position in array of pointer to input file */
#define IN_FILE_ARG 1        /* argument number of input file name */
#define OUT_FILE_NAME 1      /* position in array of pointer to output file */
#define OUT_FILE_ARG 2       /* argument number of output file name */
#define FILTER_SIZE_ARG 3    /* agrument number for the position of the filter window */
#define HOW_MANY_FILES 2     /* how many files the program opens */
#define ADDR_LENGTH 4095     /* 2**12   1, 4096 elements for 12 bit addresses */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]);

void main(int argc, char *argv[])
{
    FILE    **fname;         /* array of pointers to files */

    int     f_cnt   0,
            mlp,
            lp,
            newlp,
            oldlp,
            size,            /* size of the filter window from command line */
            hsize,           /* size integer divided by 2 */
                             /* NOTE: C always rounds toward zero, 5/2=2, etc. */
            out_count = 0,
            clos_tst,
            clos_err = 0;

    long int    *addr,
                *filt,
                loc,
                temp;        /* be careful that this dosen't overflow */

    char    msg[10];
    void    usage( char *pgrm_name[] ); /* function */

/*********************************************************************/


        /* very basic command line check, mainly to print usage if no args given */
    if( argc < HOW_MANY_ARGS ){
        usage( argv );
        exit(3);
        }

        /* read in the size of the filter window */
    size   atoi(argv[FILTER_SIZE_ARG]);
    hsize = size / 2;                       /* integer arguments, therefor integer divide */

        /* check to make sure that the filter length is odd, the algorithm depends on this */
    if( (size  % 2) == 0){
        fprintf(stderr, "\nInvalid window size, please choose a window thats length is odd.\n\n");
        exit(10);
        }

        /* open array for the addresses */
    addr   (long int *) calloc( (ADDR_LENGTH + 1), (unsigned int) sizeof( long int ) );
    if( addr == NULL ){
```

```
    perror( "\nMEMORY ALLOCATION ERROR" );
    fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR ADDRESSES\n" );
    exit(7);
    }

    /* open array for the filter to work on */
    /* NOTE:  add size because (2 * size/2 = size) */
filt = (long int *) calloc( (ADDR_LENGTH + (size - 1) ), (unsigned int) sizeof( long int ) );
if( filt == NULL ){
    perror( "\nMEMORY ALLOCATION ERROR" );
    fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR WORKING FILTER ARRAY\n" );
    exit(7);
    }

    /* open array to store pointers to all the needed files */
fname   ( FILE ** ) calloc( (HOW_MANY_FILES), (unsigned int) sizeof( FILE * ) );
if( fname == NULL ){
    perror( "\nMEMORY ALLOCATION ERROR" );
    fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR FILES\n" );
    exit(4);
    }

    /* open input file */
if( (fname[IN_FILE_NAME]   fopen(argv[IN_FILE_ARG], "r")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING INPUT FILE:  %s\n", argv[IN_FILE_ARG]);
    perror( strcat("\nINPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(1);
    }
f_cnt += 1;

/* display the input file name to the user */
printf("\n Input file is:\t\t%s\n", argv[IN_FILE_ARG]);

    /* open output file */
if( (fname[OUT_FILE_NAME]   fopen(argv[OUT_FILE_ARG], "w")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING OUTPUT FILE:  %s\n", argv[OUT_FILE_ARG]);
    perror( strcat("\nOUTPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(3);
    }
f_cnt += 1;

/* display the output file name to the user */
printf("\n Output file is:\t%s\n\n", argv[OUT_FILE_ARG]);

/* display message to acknowledge that something is happening */
fprintf( stderr, "\nReading values into array.\n");
fflush( stderr );

    /* read first string from the input file */
if (NULL == fgets(msg, 10, fname[IN_FILE_NAME]) ){
    printf("\n\t ERROR:  no data in file!\n\n");
    exit(1);
    }
else{
    /* this section is where addresses are read into an array */
    loc = 0;
    do {
          /* to prevent improper size input files from trashing memory */
        addr[loc % (ADDR_LENGTH + 1)] = atoi(msg);
        loc += 1;
        }
        while (NULL != fgets(msg, 10, fname[IN_FILE_NAME]) );
    }
```

```
if (loc > 4096){
    fprintf(stdout, "NOTE:  input file too long, aliasing has occurred.\n"
                    "Input file contained %ld values.\n", loc);
    }
    else if (loc < 4096){
        fprintf(stdout, "NOTE:  not enough values in input file.\n"
                        "Input file contained %ld values.\n", loc);
        }
    else{
        fprintf(stdout, "%ld values read into previous code array.\n", loc);
        }

    /* move addr array to filt array, and then pad ends to allow filter to operate properly */
for(oldlp = 0, newlp = (0 + hsize); oldlp <= ADDR_LENGTH; oldlp++, newlp++){
    filt[newlp] = addr[oldlp];
    }
for(newlp = 0; newlp <= (hsize   1); newlp++){
    filt[newlp]   addr[0];
    }
for(newlp   (ADDR_LENGTH + hsize + 1); newlp <= (ADDR_LENGTH + size   1); newlp++){
    filt[newlp]  : addr[ADDR_LENGTH];
    }

    /* the next section does all the work, it implements a convolution to smooth the data */
for(lp = hsize; lp <= (ADDR_LENGTH + hsize); lp++){
    temp = 0;
    for(newlp   (lp   hsize); newlp <= (lp + hsize); newlp++){
            /* beware of overflow here */
            /* that should only be possible if an extremely large number of symbols are */
            /* used to generate the address data (extremely large) */
        temp += filt[newlp];
        }
    addr[lp - hsize]   (temp + hsize) / size;   /* add hsize in to aid rounding */
    }

    /* end of convolution section */

for (lp   0; lp <= (ADDR_LENGTH   1); lp++){
    /* need to delete \n from last symbol to prevent downstream errors */
    out_count = fprintf (fname[OUT_FILE_NAME], "%d\n", addr[lp]);
    if( out_count == EOF ){
        perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_FILE_NAME] );
        clos_err   2;
        }
    }

/* do last output without any carriage return/line feed */
/* code was put inline for speed (could have used an if clause above) */
out_count   fprintf (fname[OUT_FILE_NAME], "%d", addr[ADDR_LENGTH]);
if( out_count == EOF ){
    perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_FILE_NAME] );
    clos_err = 2;
    }

for( mlp = 0; mlp < f_cnt; mlp++ ){
    clos_tst = fclose(fname[mlp]);
        /* EOF is -1 */
    if( clos_tst == EOF ){
        perror( strcat("FILE CLOSE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR CLOSING FILE:  %s\n", fname[mlp] );
        clos_err = 1;
        }
    }

    /* free all allocated memory */
free( addr );
free( fname );

if( clos_err == 1 )
    exit(2);
else if(clos_err == 2)
    exit(5);
```

```
    else
        exit(0);


}

/*********************************************************************/


    /* message to print if wrong number of arguments were given */
void usage( char *fargv[] )
{

fprintf( stderr, "\n\n "

"This program smooths data from an input file, \n\t "
    "and writes the data to another output file.\n\n "

    "The program works with 12 bit addresses.\n\n "

"THE CORRECT INVOCATION SYNTAX IS:\n \
%s input_file_name output_file_name smoothing_window_size\n\n", fargv[0] );

}

/*********************************************************************/
```

Appendix A4 - Source Code for addr-gen.c

This appendix contains the source code listing for the C language file addr-gen.c. The purpose of this program was to use the two hashing functions (see Sections 2.1 and 2.2 of this document for a description of these) to generate address values from the two input files representing symbols and previous codes. The program produces a separate set of output for each of the hashing functions. The resulting data was written to output files to allow further processing by other programs.

The program operated by opening several arrays. One of the arrays was filled with values from the input file representing the previous codes. The values from the file representing the symbols were then read. The first value was used to prime the previous code variable and then the remaining values were read in and used to calculate addresses. The addresses were used to increment counts in arrays for each of the hashing functions. Data in Appendix C9 was post-processed by this program, and the results are shown in the same appendix.

PROGRAM PARAMETERS

program name	
first argument	name of the input code file
second argument	name of the input symbol file
third argument	name of the output file for the modulo hashing results
fourth argument	name of the output file for the XOR hashing results

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*                                                                        */
/*                                                                        */
/*                  Copyright (C) 1992 by James Goeke                     */
/*                        All rights reserved.                            */
/*                                                                        */
/*                                                                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */


#define HOW_MANY_ARGS 5     /* how many arguments expected on command line */
#define PROGRAM_NAME 0      /* argument number of program name */
#define IN_CODE_FILE_NAME 0 /* position in array of pointer to input file */
#define IN_CODE_FILE_ARG 1  /* argument number of input file name */
#define IN_SYM_FILE_NAME 1  /* position in array of pointer to input file */
#define IN_SYM_FILE_ARG 2   /* argument number of input file name */
#define OUT_MOD_FILE_NAME 2 /* position in array of pointer to output file */
#define OUT_MOD_FILE_ARG 3  /* argument number of output file name */
#define OUT_XOR_FILE_NAME 3 /* position in array of pointer to output file */
#define OUT_XOR_FILE_ARG 4  /* argument number of output file name */
#define HOW_MANY_FILES 4    /* how many files the program opens */
#define HIST_LENGTH 4095    /* 2**12 -1, 4096 elements for 12 bit codes */
#define ADDR_MASK 0x0fff    /* mask all addresses to 12 bits */
#define TABLE_BITS 12       /* 12 bit addresses are used to access table */
#define SYMBOL_BITS 8       /* symbols are 8 bits long */
#define SYMBOL_LEN 255      /* 2**8   1 */
#define SYMBOL_MASK 0x00ff  /* to garuantee only 8 active bits */
#define MOD_VALUE 4021      /* magic number */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]);

void main(int argc, char *argv[])
{
    FILE    **fname;        /* array of pointers to files */

    int     f_cnt   0,
            mlp,
            lp,
            loc,
            out_count = 0,
            clos_tst,
            clos_err   0;

    long int    *modhist,   /* array for histogramming modulo accesses */
                *xorhist;   /* array for histogramming xor accesses */

    short unsigned int  *pcodeary,  /* array of values read in for previous codes */
                        *revary;    /* array of bit reversed patterns for all 8 bit values */

    long unsigned int   harg;

    unsigned int    pcodem, /* previous code for modulus address */
                    pcodex, /* previous code for xor address */
                    addr,
                    symbol;

    char    msg[10];
    void    usage( char *pgrm_name[] );             /* function */
    void    bit_rev_ary( short unsigned int *ary);  /* function */

/*********************************************************************/


       /* very basic command line check, mainly to print usage if no args given */
    if( argc < HOW_MANY_ARGS ){
        usage( argv );
        exit(3);
        }
```

```
        /* open array for the modulo histogram */
    modhist = (long int *) calloc( (HIST_LENGTH + 1), (unsigned int) sizeof( long int ) );
    if( modhist == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR MOD HISTOGRAM\n" );
        exit(7);
        }

        /* open array for the xor histogram */
    xorhist = (long int *) calloc( (HIST_LENGTH + 1), (unsigned int) sizeof( long int ) );
    if( xorhist == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR XOR HISTOGRAM\n" );
        exit(7);
        }

        /* open array for the previous codes */
    pcodeary = (short unsigned int *)
        calloc( (HIST_LENGTH + 1), (unsigned int) sizeof(short unsigned int ) );
    if( pcodeary == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR PREVIOUS CODE ARRAY\n" );
        exit(7);
        }

        /* open array for the reverse bit pattern of all 8 bit binary values */
    revary = (short unsigned int *)
        calloc( (SYMBOL_LEN + 1), (unsigned int) sizeof(short unsigned int ) );
    if( revary == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR REVERSE BITS ARRAY\n" );
        exit(7);
        }

        /* go ahead and build the array */
    bit_rev_ary( revary);

        /* open array to store pointers to all the needed files */
    fname = ( FILE ** ) calloc( (HOW_MANY_FILES), (unsigned int) sizeof( FILE * ) );
    if( fname == NULL ){
        perror( "\nMEMORY ALLOCATION ERROR" );
        fprintf( stderr, "\n\tERROR ALLOCATING MEMORY FOR FILES\n" );
        exit(4);
        }

        /* open previous code input file */
    if( (fname[IN_CODE_FILE_NAME] = fopen(argv[IN_CODE_FILE_ARG], "r")) == NULL ){
        fprintf( stderr, "\n\tERROR OPENING PREVIOUS CODE INPUT FILE:  %s\n",
argv[IN_CODE_FILE_ARG]);
        perror( strcat("\nINPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        exit(1);
        }
    f_cnt += 1;

    /* display the input file name to the user */
    printf("\n Previous code input file is:\t%s\n", argv[IN_CODE_FILE_ARG]);

        /* open symbol input file */
    if( (fname[IN_SYM_FILE_NAME]   fopen(argv[IN_SYM_FILE_ARG], "r")) == NULL ){
        fprintf( stderr, "\n\tERROR OPENING SYMBOL INPUT FILE:  %s\n", argv[IN_SYM_FILE_ARG]);
        perror( strcat("\nINPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        exit(1);
        }
    f_cnt += 1;

    /* display the input file name to the user */
    printf(" Symbol input file is:\t\t%s\n", argv[IN_SYM_FILE_ARG]);
```

```
    /* open modulo output file */
if( (fname[OUT_MOD_FILE_NAME]   fopen(argv[OUT_MOD_FILE_ARG], "w")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING MODULO OUTPUT FILE:  %s\n", argv[OUT_MOD_FILE_ARG]);
    perror( strcat("\nOUTPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(3);
    }
f_cnt += 1;

/* display the modulo output file name to the user */
printf(" Output modulo file is:\t\t%s\n", argv[OUT_MOD_FILE_ARG]);

    /* open xor output file */
if( (fname[OUT_XOR_FILE_NAME]   fopen(argv[OUT_XOR_FILE_ARG], "w")) == NULL ){
    fprintf( stderr, "\n\tERROR OPENING OUTPUT FILE:  %s\n", argv[OUT_XOR_FILE_ARG]);
    perror( strcat("\nOUTPUT FILE OPEN ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    exit(3);
    }
f_cnt += 1;

/* display the xor output file name to the user */
printf(" Output xor file is:\t\t%s\n\n", argv[OUT_XOR_FILE_ARG]);

/* display message to acknowledge that something is happening */
fprintf( stderr, "\nReading previous codes into array.\n");
fflush( stderr );

    /* read first string from the input file */
if (NULL == fgets(msg, 10, fname[IN_CODE_FILE_NAME]) ){
    printf("\n\t ERROR:  no data in file!\n\n");
    exit(1);
    }
else{
    /* this section is where previous codes are read into an array */
    loc = 0;
    do {
            /* to prevent improper size input files from trashing memory */
        pcodeary[loc % (HIST_LENGTH + 1)] = atoi(msg);
        loc += 1;
        }
        while (NULL != fgets(msg, 10, fname[IN_CODE_FILE_NAME]) );
    }

if (loc > 4096){
    fprintf(stdout, "NOTE:  previous code input file too long, aliasing has occurred.\n"
                    "Input file contained %d values.\n", loc);
    }
    else if (loc < 4096){
        fprintf(stdout, "ERROR:  not enough values in previous code input file.\n"
                        "Input file contained %d values.\n", loc);
        }
    else{
        fprintf(stdout, "%d values read into previous code array.\n", loc);
        }

/* display message to acknowledge that something is happening */
fprintf( stderr, "\nGenerating the address data.\n");
fflush( stderr );
```

```
    /* read first string from the input file */
if (NULL == fgets(msg, 10, fname[IN_SYM_FILE_NAME]) ){
    printf("\n\t ERROR:  no data in file!\n\n");
    exit(1);
    }
else{
    /* this section is where all the addresses are generated */
    /* NOTE:  the first symbol read will be used as the prime for */
    /* the previous code, the second symbol read will become the */
    /* first symbol. */
    /* NOTE:  after the first time through each address generator uses */
    /* the previous code derived as a result of its address calculation */
    pcodem = atoi(msg);
    pcodex = pcodem;
    fgets(msg, 10, fname[IN_SYM_FILE_NAME]);
    do {
        symbol = atoi(msg);
            /* generate the modulo addresses */
        addr = (ADDR_MASK) & (((symbol << TABLE_BITS) + pcodem) % MOD_VALUE);
        modhist[addr] += 1;
        pcodem = pcodeary[addr];
            /* generate the xor addresses */
        harg = ( ( ((long)(symbol ^ revary[symbol])) << TABLE_BITS) + pcodex );
        addr = (ADDR_MASK) & (unsigned int)(( harg >> (SYMBOL_BITS)) ^ pcodex);
        xorhist[addr] += 1;
        pcodex = pcodeary[addr];
        }
        while (NULL != fgets(msg, 10, fname[IN_SYM_FILE_NAME]) );
    }

    /* write histogram values out to memory */
    /* write out values from 0 to last element minus one (see next code blocks) */
for (lp = 0; lp <= (HIST_LENGTH   1); lp++){
        /* need to delete \n from last symbol to prevent downstream errors */
        /* write modulo histogram to its file */
    out_count = fprintf (fname[OUT_MOD_FILE_NAME], "%d\n", modhist[lp]);
    if( out_count == EOF ){
        perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_MOD_FILE_NAME] );
        clos_err   2;
        }
        /* write xor histogram to its file */
    out_count = fprintf (fname[OUT_XOR_FILE_NAME], "%d\n", xorhist[lp]);
    if( out_count == EOF ){
        perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
        fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_XOR_FILE_NAME] );
        clos_err - 2;
        }
    }

    /* do last modulo output without any carriage return/line feed */
    /* code was put inline for speed (could have used an if clause above) */
    /* write out last element of the array */
out_count = fprintf (fname[OUT_MOD_FILE_NAME], "%d", modhist[HIST_LENGTH]);
if( out_count == EOF ){
    perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_MOD_FILE_NAME] );
    clos_err   2;
    }

    /* do last xor output without any carriage return/line feed */
    /* code was put inline for speed (could have used an if clause above) */
    /* write out last element of the array */
out_count   fprintf (fname[OUT_XOR_FILE_NAME], "%d", xorhist[HIST_LENGTH]);
if( out_count == EOF ){
    perror( strcat("FILE WRITE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
    fprintf( stderr, "\n\tERROR WRITING FILE:  %s\n", fname[OUT_XOR_FILE_NAME] );
    clos_err = 2;
    }

for( mlp   0; mlp < f_cnt; mlp++ ){
clos_tst   fclose( fname[mlp] );
if( clos_tst == EOF ){
    perror( strcat("FILE CLOSE ERROR IN PROGRAM ", argv[PROGRAM_NAME]) );
```

```
        fprintf( stderr, "\n\tERROR CLOSING FILE:  %s\n", fname[mlp] );
        clos_err   1;
        }
    }

        /* free all allocated memory */
    free( modhist );
    free( xorhist );
    free( pcodeary );
    free( revary );
    free( fname );

    if( clos_err == 1 )
        exit(2);
    else if(clos_err == 2)
        exit(5);
    else
        exit(0);


}


/*********************************************************************/


    /* message to print if wrong number of arguments were given */
void usage( char *fargv[] )
{

fprintf( stderr, "\n\n"

"This program histograms data generated by mod and xor operations.\n"
"A file containing 4096 'previous codes' and another containing 'symbols' are read.\n"
"These values are used to generate simulated addresses.\n\n"

    "The program works with 12 bit addresses, so 4096 bins are used.\n\n "

"THE CORRECT INVOCATION SYNTAX IS:\n \
%s\n\t\
input_code_file_name input_symbol_file_name\n\t\
output_mod_file_name output_xor_file_name\n\n", fargv[0] );

}


/*********************************************************************/
```

```
    /* function to produce a reverse bit pattern for all 8 bit values */
void bit_rev_ary( short unsigned int *stor_ary )
{

    struct bit_fields
    {
    unsigned short int bit0 : 1;
    unsigned short int bit1 : 1;
    unsigned short int bit2 : 1;
    unsigned short int bit3 : 1;
    unsigned short int bit4 : 1;
    unsigned short int bit5 : 1;
    unsigned short int bit6 : 1;
    unsigned short int bit7 : 1;
    };

    union symbol
    {
    unsigned short int value;
    struct bit_fields bits;
    };

    union symbol in_sym, out_sym;
    int lp;

    for(lp = 0; lp <= 255; lp++)
        {
         in_sym.value = lp;

         out_sym.bits.bit7   in_sym.bits.bit0;
         out_sym.bits.bit6   in_sym.bits.bit1;
         out_sym.bits.bit5   in_sym.bits.bit2;
         out_sym.bits.bit4   in_sym.bits.bit3;
         out_sym.bits.bit3   in_sym.bits.bit4;
         out_sym.bits.bit2   in_sym.bits.bit5;
         out_sym.bits.bit1   in_sym.bits.bit6;
         out_sym.bits.bit0   in_sym.bits.bit7;

         stor_ary[lp] = (SYMBOL_MASK) & (out_sym.value);
         }

}


/*********************************************************************/
```

APPENDIX B - Input Data Sets

The following table lists the different data sets and how each of them were produced. The left column also references the sub-appendices that will display the Mathcad documents used to generate and/or analyze the data.

Input Data Sets					
Directory (Appendix)	Random Values Produced				
	How Many	Range	Distribution	Seed Value	Method
CDEUNIF (B1)	4096	0-4095 12 bits	Uniform	5	Mathcad
SYMUNIF (B2)	4096	0-255 8 bits	Uniform	5	Mathcad
CDEGAUSS (B3)	4096	0-4095 12 bits	Gaussian	5	Mathcad
SYMGAUSS (B4)	4096	0-255 8 bits	Gaussian	5	Mathcad
CDEMYC1 (B5)	4096	0-4095 12 bits	Random	5	make-sym.c
SYMMYC1 (B6)	65536	0-255 8 bits	Random	5	make-sym.c

Appendix B1 Input Data Set CDEUNIF

This appendix contains the two Mathcad documents used to generate and check the data used to produce previous codes that are uniformly distributed random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named CDEUNIF.

The first Mathcad document, Figure B1-1, is the document used to produced a sequence of random numbers and write them to a file. Only two pieces of information are needed for this. How many values to produce, and the range into which these values should fall. For this set of data 4096 values were produced in the range from 0 to 4095.

The second Mathcad document, Figure B1-2, is the document that was used to check the data produced by the previous document after processing by the C language program histogra.c. It reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. Because of the size of the plot and the number of data points the plot is very busy, so it is also displayed in the lower plot as a scatter plot. Looking at the scatter plot it is clear that only integer values are being used. The data is randomly scattered in the x direction, but in the y direction all the points fall on integers. In addition, at the very top of Figure B1-2 some parameters of the data have been calculated and displayed.

Data generation. This is file GEN.MCD.

This range defines how many values to generate.

count := 0..4095 Produce 4096 values.

This variable defines the range of values to produce.

value := 4096

Equation to define the distribution of values.

$output_{count}$:= floor(rnd(value)) This will produce values in the range from 0 to 4095.

NOTE: the Mathcad rnd function is designed to produce uniformally distributed values in the range determined by the parameter passed to the function.

WRITEPRN(code) := output This writes the calculated values out to disk.

THE SEED VALUE USED WAS 5.

The Mathcad rnd function uses an incrementing seed value. Everytime that the function is calculated the seed is incremented to a new value so the next iteration will produce different results. To be able to reproduce a series of values a particular seed must be used. This is accomplished by selecting the randomize entry on the math pulldown menu. This allows you to select a seed to be used for the next iteration. Be careful however, as this is only good for a single iteration after which the seed will again be incremented.

Figure B1-1 - Data set CDEUNIF: Mathcad Document GEN.MCD

Input data display. This is file HIST.MCD.

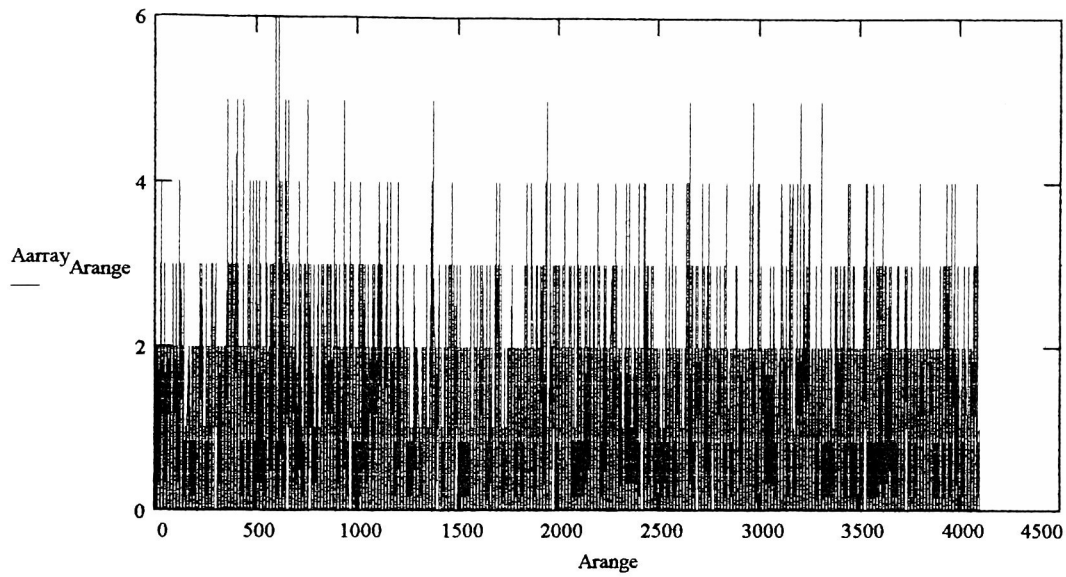
Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin := min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd := stdev(Aarray)
Amin = 0	Amax = 6	Amean = 1	Astd = 1.0066

Display array. Arange := 0 .. last(Aarray)



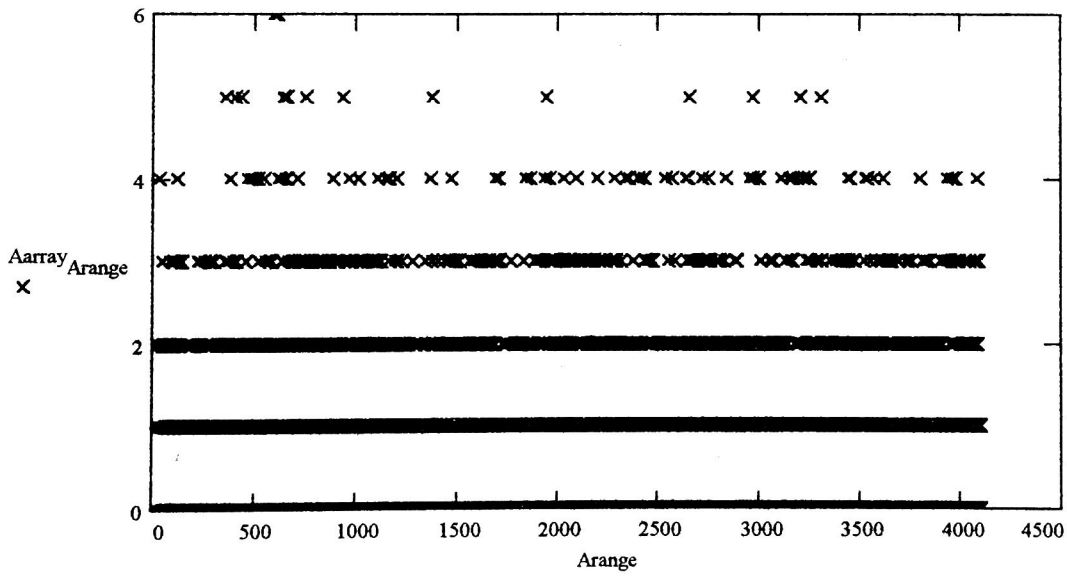


Figure B1-2 - Data set CDEUNIF: Mathcad Document HIST.MCD

Appendix B2 Input Data Set SYMUNIF

This appendix contains the two Mathcad documents used to generate and check the data used to produce symbols that are uniformly distributed random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named SYMUNIF.

The first Mathcad document, Figure B2-1, is the document used to produced a sequence of random numbers and write them to a file. Only two pieces of information are needed for this. How many values to produce, and the range into which these values should fall. For this set of data 4096 values were produced in the range from 0 to 255.

The second Mathcad document, Figure B2-2, is the document that was used to check the data produced by the previous document after processing by the C language program histogra.c. It reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. The data is also shown as a scatter plot in the lower plot. Looking at this plot it is not as clear that only integer values are being used as in the previous appendix This is mainly because of the scale on the y axis, but it still can be seen that the y values only fall on integers. In addition, at the very top of Figure B2-2 some parameters of the data have been calculated and displayed.

Data generation. This is file GEN.MCD.

This range defines how many values to generate.

count := 0..4095 Produce 4096 values.

This variable defines the range of values to produce.

value := 256

Equation to define the distribution of values.

$output_{count}$ floor(rnd(value)) This will produce values in the range from 0 to 4095.

NOTE: the Mathcad rnd function is designed to produce uniformally distributed values in the range determined by the parameter passed to the function.

WRITEPRN(code) := output This writes the calculated values out to disk.

THE SEED VALUE USED WAS 5.

The Mathcad rnd function uses an incrementing seed value. Everytime that the function is calculated the seed is incremented to a new value so the next iteration will produce different results. To be able to reproduce a series of values a particular seed must be used. This is accomplished by selecting the randomize entry on the math pulldown menu. This allows you to select a seed to be used for the next iteration. Be careful however, as this is only good for a single iteration after which the seed will again be incremented.

Figure B2-1 Input Data Set SYMUNIF: Mathcad Document GEN.MCD

Input data display. This is file HIST.MCD.

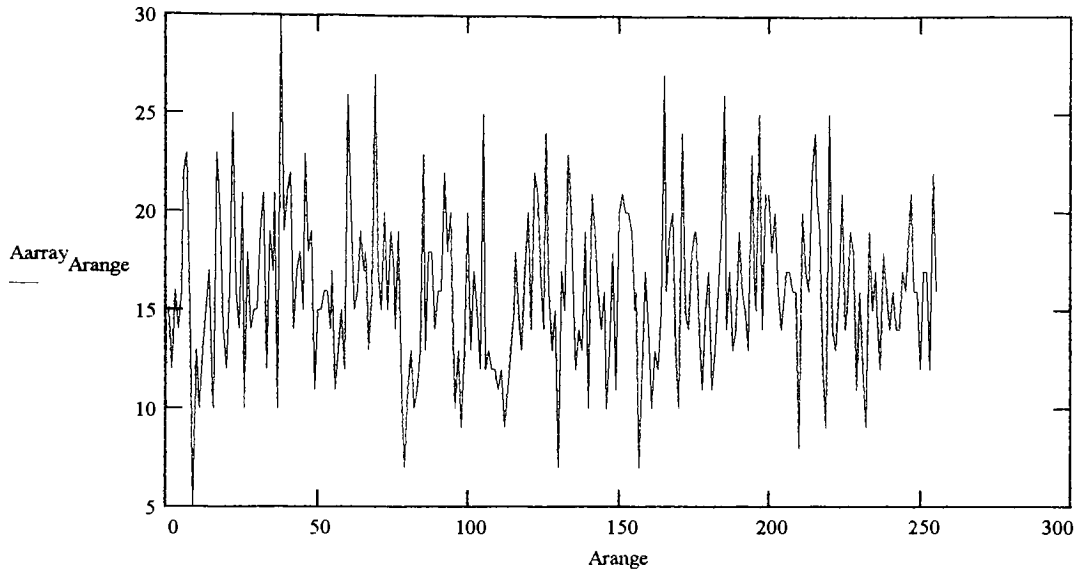
Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin = min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd = stdev(Aarray)
Amin = 5	Amax = 30	Amean = 16	Astd = 4.1022

Display array. Arange := 0.. last(Aarray)



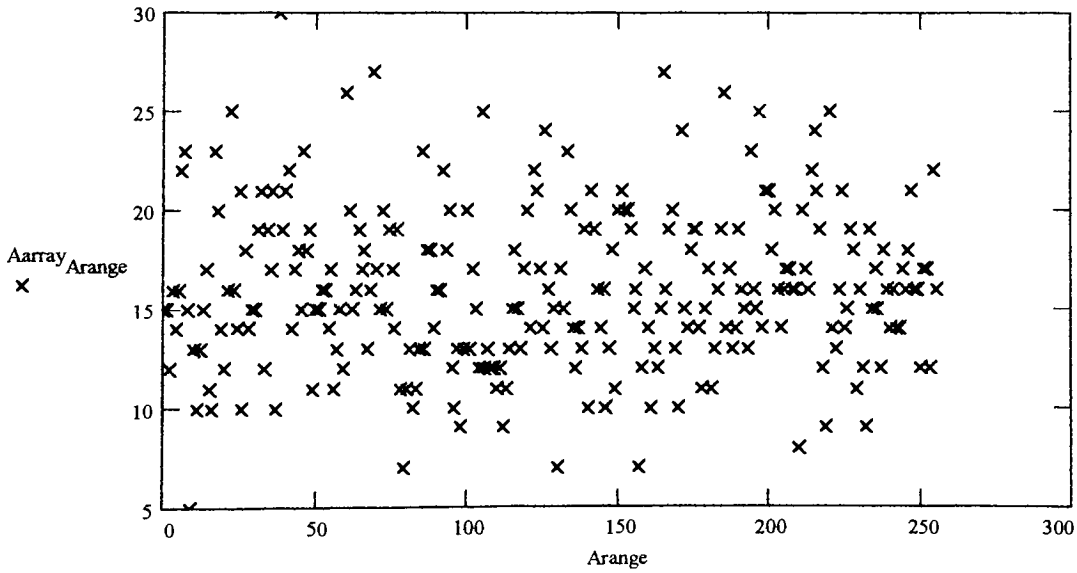


Figure B2-2 - Data set SYMUNIF: Mathcad Document HIST.MCD

Appendix B3 - Input Data Set CDEGAUSS

This appendix contains the two Mathcad documents used to generate and check the data used to produce previous codes that have a gaussian distribution of random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named CDEGAUSS.

The first Mathcad document, Figure B3-1, is the document used to produced a sequence of random numbers and write them to a file. To produce this distribution four pieces of information are needed. How many values to produce, the range into which these values should fall, the mean of the data set, and a standard deviation. For this set of data 4096 values were produced in the range from 0 to 4095.

The second Mathcad document, Figure B3-2, is the document that was used to check the data produced by the previous document after processing by the C language program histogra.c. It reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. The data is also shown as a scatter plot in the lower plot. Looking at this plot it is clear that only integer values are being used. The data is randomly scattered in the x direction, but in the y direction all the points fall on integers. Looking at either plot it is clearly shown the data has a gaussian distribution. In addition, at the very top of Figure B3-2 some parameters of the data have been calculated and displayed.

Data generation. This is file GEN.MCD.

This range defines how many values to generate.

count := 0..4095 Produce 4096 values.

This variable defines the range of values to produce.

value := 4096

These variables define the parameters the control the distribution of the values produced.

$$mean\_pt := \frac{value}{2}$$

$$std\_pt := \frac{value}{16}$$

Equation to define the distribution of values.

$$output_{count} := floor\left[mean\_pt + \left(std\_pt \cdot \sqrt{-2 \cdot \ln(md(1))}\right) \cdot \cos(2 \cdot \pi \cdot rnd(1))\right]$$

This equation comes from the Mathcad handbook. It produces a gaussian distribution of values that is centered at 'mean_pt' and has a spread of 'std_pt'.

WRITEPRN(code) := output This writes the calculated values out to disk.

THE SEED VALUE USED WAS 5.

The Mathcad rnd function uses an incrementing seed value. Everytime that the function is calculated the seed is incremented to a new value so the next iteration will produce different results. To be able to reproduce a series of values a particular seed must be used. This is accomplished by selecting the randomize entry on the math pulldown menu. This allows you to select a seed to be used for the next iteration. Be careful however, as this is only good for a single iteration after which the seed will again be incremented.

Figure B3-1 Input Data Set CDEGAUSS: Mathcad Document GEN.MCD

Input data display. This is file HIST.MCD.

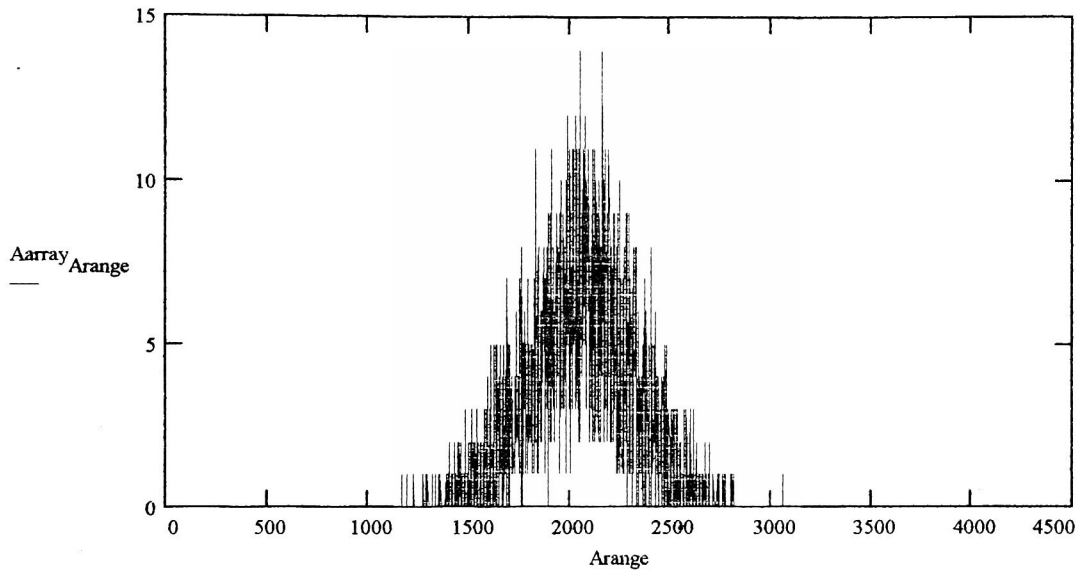
Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin := min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd := stdev(Aarray)
Amin = 0	Amax = 14	Amean = 1	Astd = 2.1473

Display array. Arange := 0 .. last(Aarray)



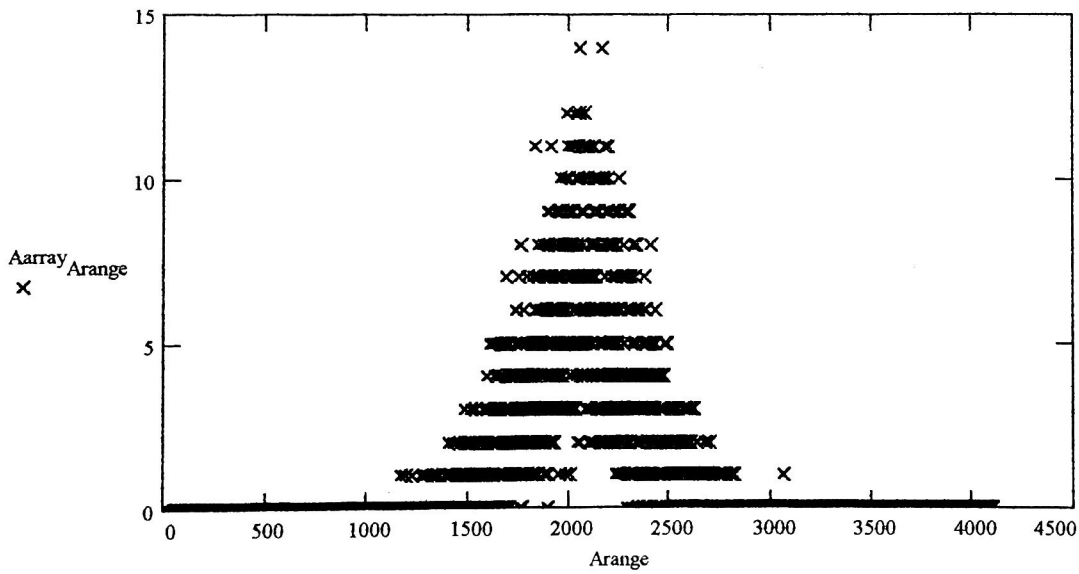


Figure B3-2 - Data set CDEGAUSS: Mathcad Document HIST.MCD

Appendix B4 Input Data Set SYMGAUSS

This appendix contains the two Mathcad documents used to generate and check the data used to produce symbols that have a gaussian distribution of random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named SYMGAUSS.

The first Mathcad document, Figure B4-1, is the document used to produced a sequence of random numbers and write them to a file. To produce this distribution four pieces of information are needed. How many values to produce, the range into which these values should fall, the mean of the data set, and a standard deviation. For this set of data 4096 values were produced in the range from 0 to 255.

The second Mathcad document, Figure B4-2, is the document that was used to check the data produced by the previous document after processing by the C language program histogra.c. It reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. The data is also shown as a scatter plot in the lower plot. Looking at this plot it is not as clear that only integer values are being used as in the previous appendix This is mainly because of the scale on the y axis. Looking at either plot it is clearly shown the data has a gaussian distribution. In addition, at the very top of Figure B4-2 some parameters of the data have been calculated and displayed.

Data generation. This is file GEN.MCD.

This range defines how many values to generate.

count := 0 .. 4095 Produce 4096 values.

This variable defines the range of values to produce.

value := 256

These variables define the parameters the control the distribution of the values produced.

$$mean\_pt := \frac{value}{2}$$

$$std\_pt := \frac{value}{16}$$

Equation to define the distribution of values.

$$output_{count} := floor\left[mean\_pt + \left(std\_pt \cdot \sqrt{-2 \cdot \ln(rnd(1))}\right) \cdot \cos(2 \cdot \pi \cdot rnd(1))\right]$$

This equation comes from the Mathcad handbook. It produces a gaussian distribution of values that is centered at 'mean_pt' and has a spread of 'std_pt'.

WRITEPRN(code) := output This writes the calculated values out to disk.

THE SEED VALUE USED WAS 5.

The Mathcad rnd function uses an incrementing seed value. Everytime that the function is calculated the seed is incremented to a new value so the next iteration will produce different results. To be able to reproduce a series of values a particular seed must be used. This is accomplished by selecting the randomize entry on the math pulldown menu. This allows you to select a seed to be used for the next iteration. Be careful however, as this is only good for a single iteration after which the seed will again be incremented.

Figure B4-1 - Input Data Set SYMGAUSS: Mathcad Document GEN.MCD

Input data display. This is file HIST.MCD.

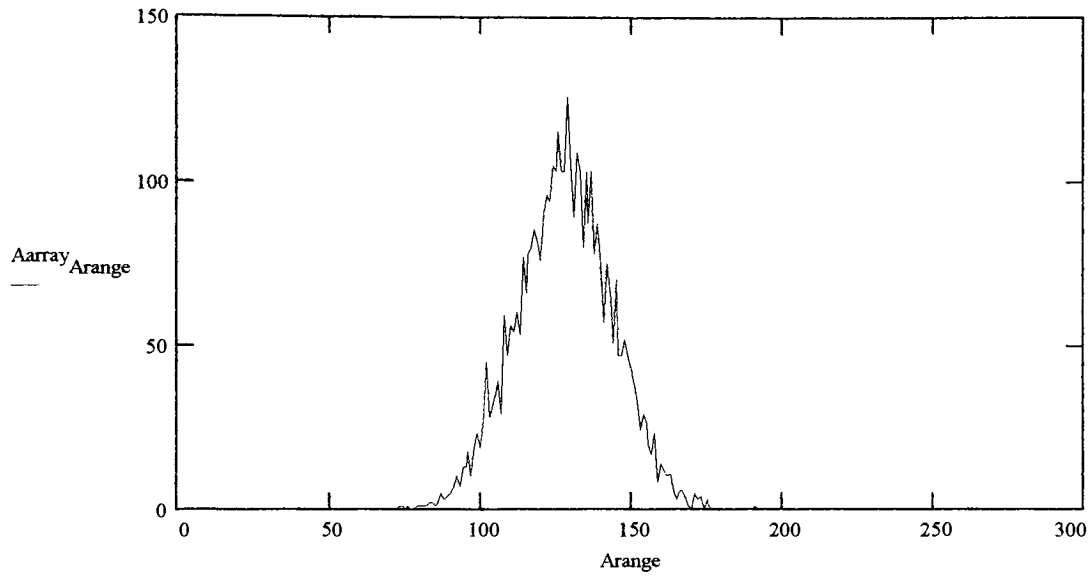
Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin := min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd := stdev(Aarray)
Amin = 0	Amax = 126	Amean = 16	Astd = 30.5124

Display array. Arange := 0.. last(Aarray)



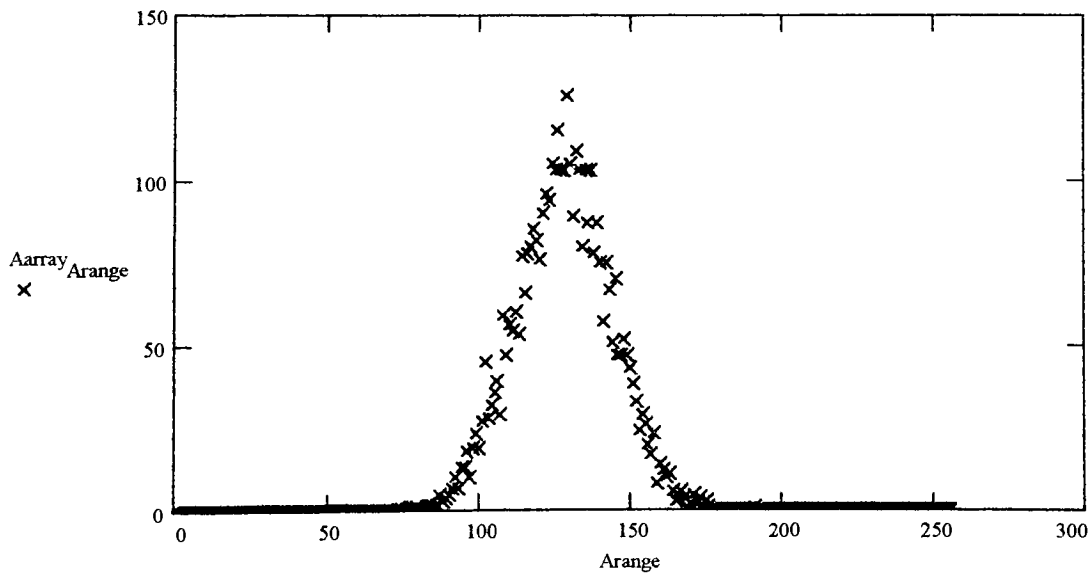


Figure B4-2 - Data set SYMGAUSS: Mathcad Document HIST.MCD

Appendix B5 Input Data Set CDEMYC1

This appendix contains the a Mathcad document used to check the data that was produced as previous codes using my C language program to produce random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named CDEMYC1.

The data was produced with a C language program to attempt to get a more "random" distribution of data. A particular distribution was not wanted. This was to get another type of data to compare and contrast with the data produced by the Mathcad documents. For this set of data 4096 values were produced in the range from 0 to 4095. The program used, make-sym.c is shown in appendix A1. The command line used was:

make-sym.exe 4096 code.prn 5 c.

The Mathcad document shown in Figure B5-1 is the document that was used to check the data produced by make-sym.c after processing by the C language program histogra.c. It just reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. Because of the size of the plot and the number of data points the plot is very busy, so it is also displayed in the lower plot as a scatter plot. Looking at this scatter plot it is clear that only integer values are being used. The data is randomly scattered in the x direction, but in the y direction all the points fall on integers. In addition, at the very top of Figure B5-1 some parameters of the data have been calculated and displayed.

Input data display. This is file HIST.MCD.

Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin := min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd := stdev(Aarray)
Amin = 0	Amax = 6	Amean = 1	Astd = 0.9778

Display array. Arange := 0 .. last(Aarray)

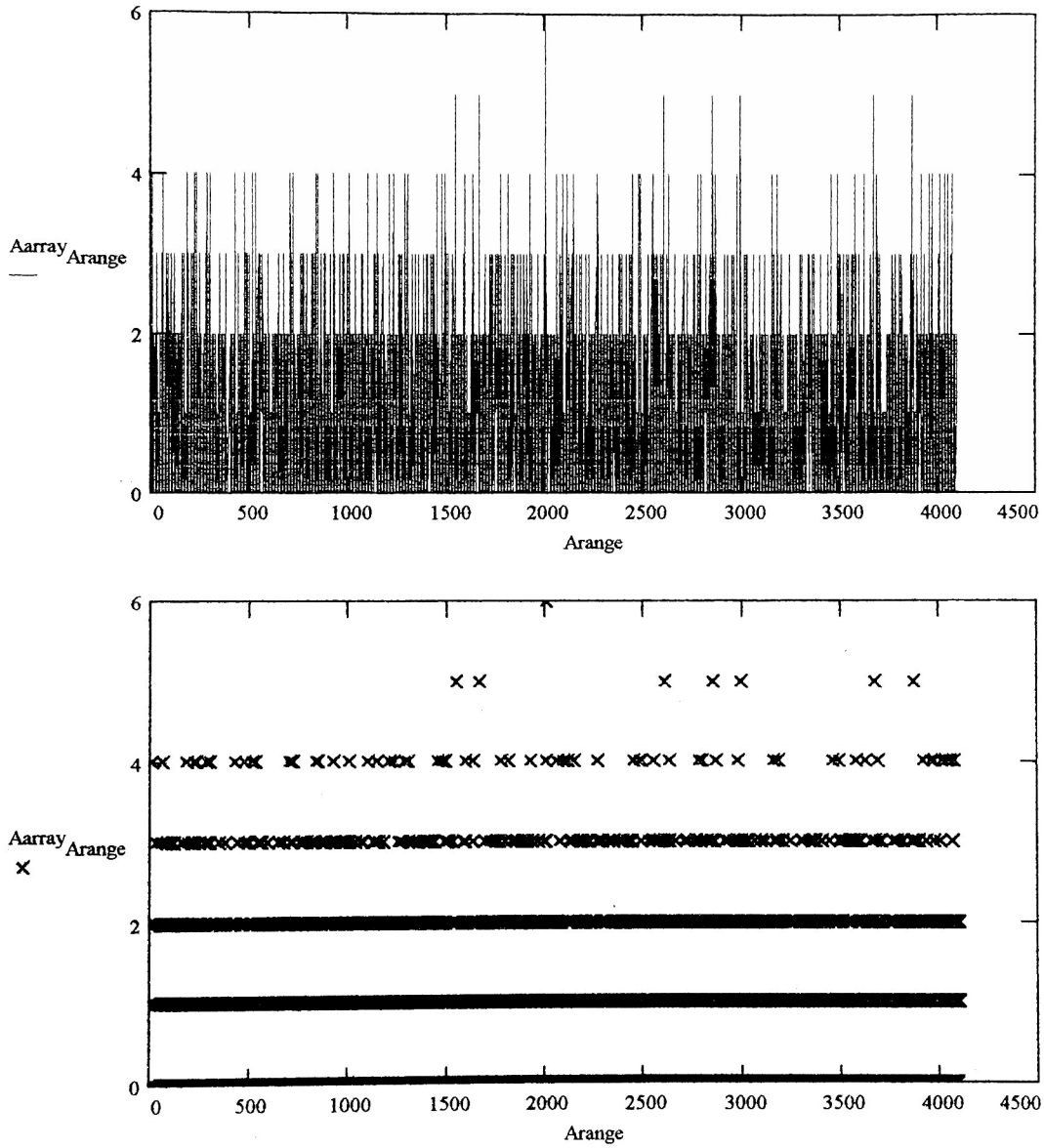


Figure B5-1 - Data set CDEMYC1: Mathcad Document GEN.MCD

Appendix B6 Input Data Set SYMMYC1

This appendix contains the a Mathcad document used to check the data that was produced as symbols using my C language program to produce random data. Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named SYMMYC1.

The data was produced with a C language program to attempt to get a more "random" distribution of data. A particular distribution was not wanted. This was to get another type of data to compare and contrast with the data produced by the Mathcad documents. For this set of data 65536 values were produced in the range from 0 to 255. The program used, make-sym.c is shown in appendix A1 The command line used was:

make-sym.exe 65536 code.prn 5 s.

The Mathcad document shown in Figure B6-1 is the document that was used to check the data produced by make-sym.c after processing by the C language program histogra.c. It reads the data in, calculates a number of parameters, and plots the data. The data was plotted in two ways. The upper plot in the figure is a line plot. The data is also shown as a scatter plot in the lower plot. Looking at this plot it is not as clear that only integer values are being used as in the previous appendix This is mainly because of the scale on the y axis. In addition, at the very top of Figure B6-1 some parameters of the data have been calculated and displayed.

Input data display. This is file HIST.MCD.

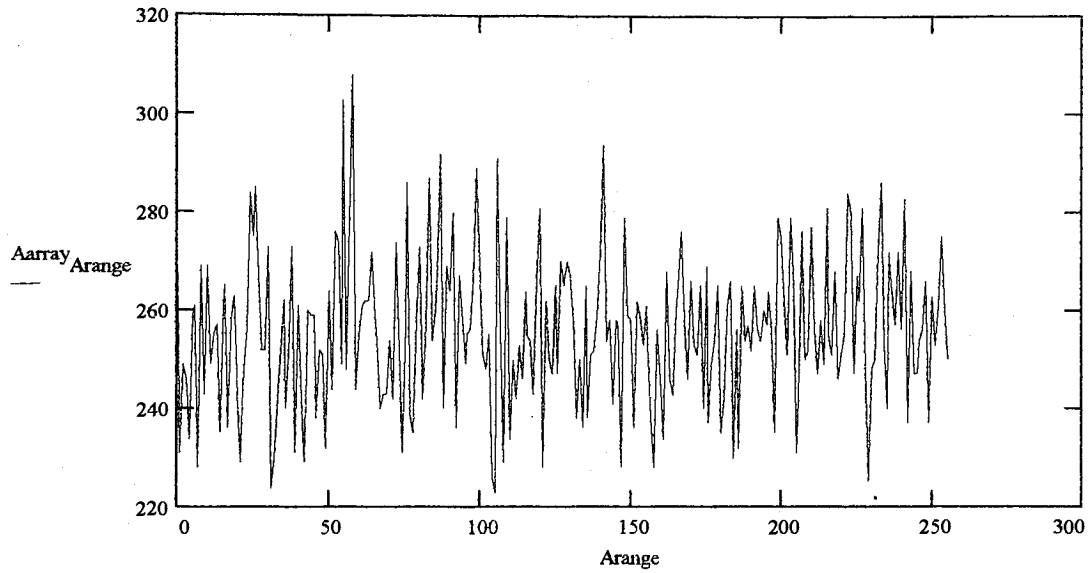
Read in the input data.

Aarray := READPRN(hist1)

Check array parameters.

Amin := min(Aarray)	Amax := max(Aarray)	Amean := mean(Aarray)	Astd := stdev(Aarray)
Amin = 223	Amax = 308	Amean = 256	Astd = 15.4067

Display array. Arange := 0 .. last(Aarray)



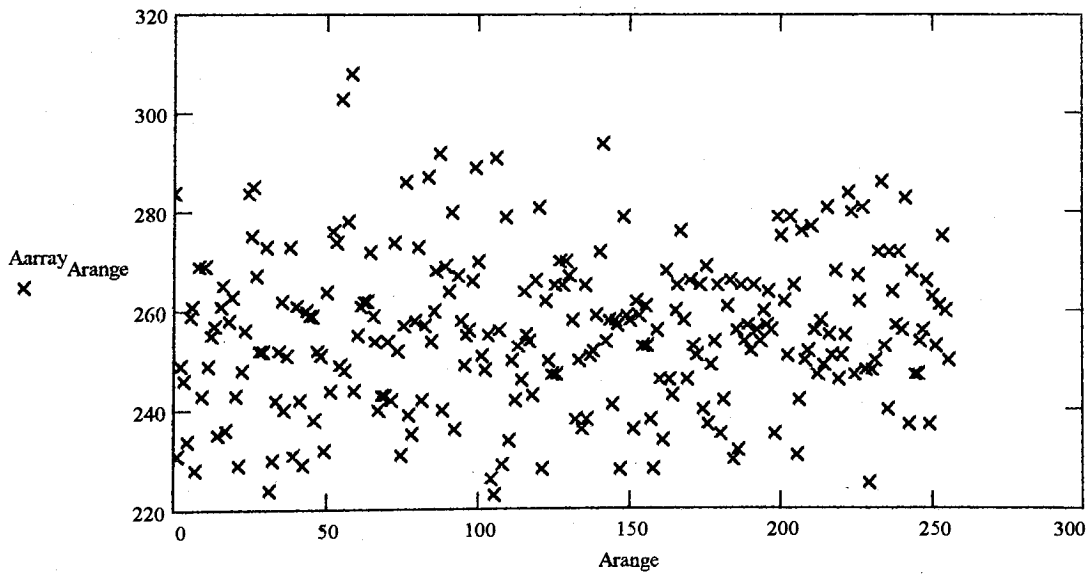


Figure B6-1 - Data set SYMMYC1: Mathcad Document GEN.MCD

APPENDIX C - Output Data Sets

The following table lists the different result data sets. The data in these sets represents using the hashing algorithms on the input files listed at the top and left of the table. The program addr-gen.c (Appendix A4) was used to generate the data in each of the directories listed in the table. The body of the table also references the sub-appendices that will display the Mathcad documents used to analyze the data.

Output Data Sets (See Appendix)		Code File		
		CDEUNIF	CDEGAUSS	CDEMYC1
Symbol File	SYMUNIF	ADDRUCUS (C1)	ADDRGCUS (C2)	ADDRMCUS (C3)
	SYMGAUSS	ADDRUCGS (C4)	ADDRGCGS (C5)	ADDRMCGS (C6)
	SYMMYC1	ADDRUCMS (C7)	ADDRGCMS (C8)	ADDRMCMS (C9)

Appendix C1 Output Data Set ADDRUCUS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the uniformly distributed previous codes (CDEUNIF) and the uniformly distributed symbols (SYMUNIF). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRUCUS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdeunif\code.prn symunif\code.prn addrucus\mod.prn addrucus\xor.prn.

The portion of the Mathcad document shown in Figure C1-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C1-2 show the unprocessed data as scatter plots. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of the upper half of a gaussian distribution. Only half a distribution is generated because of the small size of the input data set. This is more clearly seen in several of the data sets that used a larger input symbol set.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 7$	$mmean = 1$	$mstd = 1.026$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 0$	$xmax = 6$	$xmean = 1$	$xstd = 1.021$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 7$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C1-1 - Output Data Set ADDRUCUS: Mathcad Document ADDR.MCD

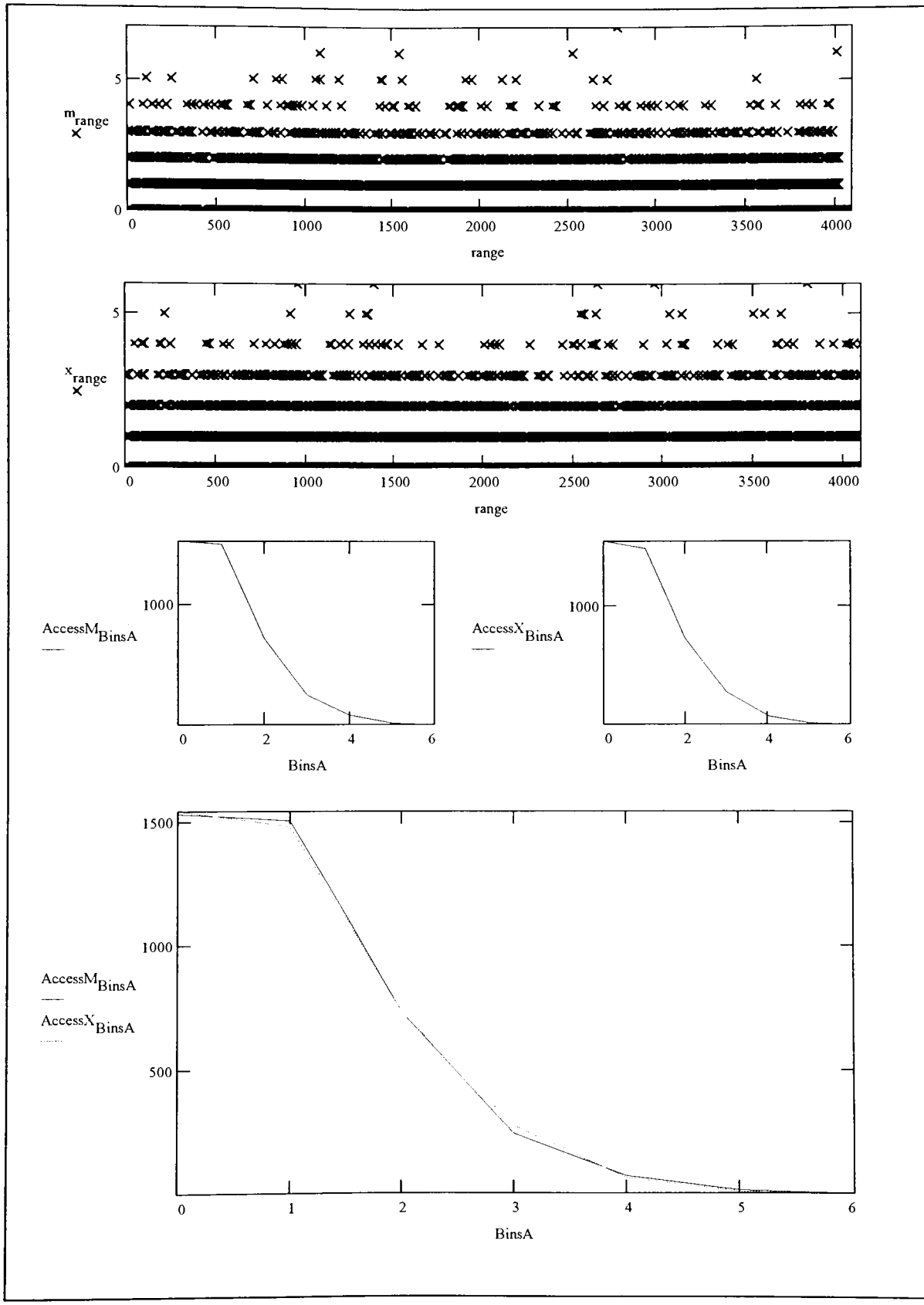


Figure C1-2 - Output Data Set ADDRUCUS: Mathcad Document ADDR.MCD

Appendix C2 - Output Data Set ADDRGCUS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the gaussian distributed previous codes (CDEGAUSS) and the uniformly distributed symbols (SYMUNIF). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRGCUS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdegauss\code.prn symunif\code.prn addrgcus\mod.prn addrgcus\xor.prn.

The portion of the Mathcad document shown in Figure C2-1 was used to check the data produced by the addr-gen.c. The upper two plots in Figure C2-2 show the unprocessed data as scatter plots, which show that the modulo algorithm produces very different results than the XOR algorithm. This is a direct result of the previous code distribution. Since it is gaussian there are a relatively small number of the possible previous codes used, but the XOR algorithm does a better job of distributing the addresses throughout the memory. The scatter plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. These plots show a highly skewed distribution because of the small size of the input data set (to see the difference between the XOR and the modulo algorithms see appendix 8 which uses a larger input data set).

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 11$	$mmean = 1$	$mstd = 1.71$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 0$	$xmax = 7$	$xmean = 1$	$xstd = 1.036$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 11$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C2-1 - Output Data Set ADDRGCUS: Mathcad Document ADDR.MCD

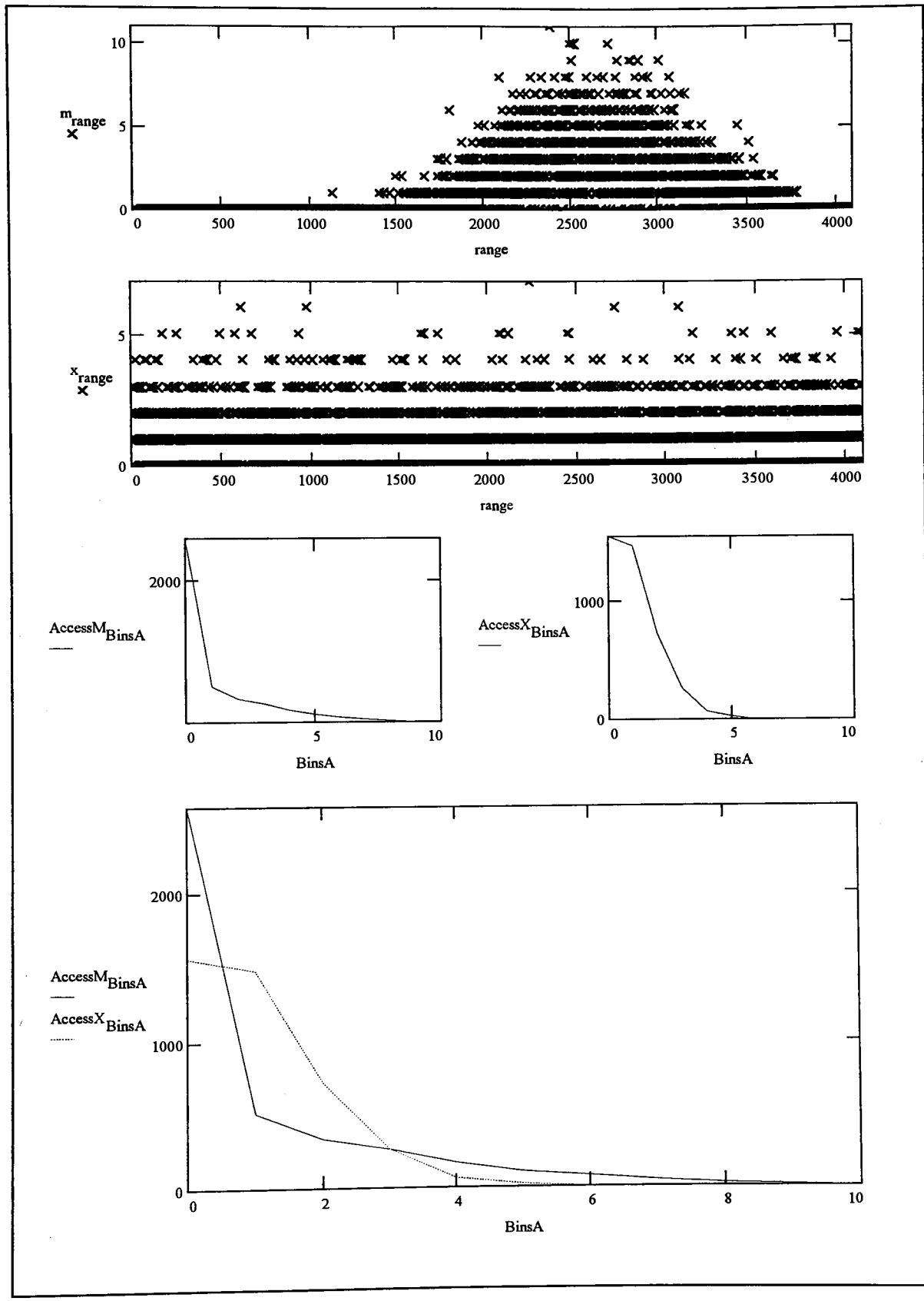


Figure C2-2 - Output Data Set ADDRGCUS: Mathcad Document ADDR.MCD

Appendix C3 Output Data Set ADDRMCUS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the my C program distributed previous codes (CDEMYC1) and the uniformly distributed symbols (SYMUNIF). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRMCUS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contain. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdemyc1\code.prn symunif\code.prn addrmcus\mod.prn addrmcus\xor.prn.

The portion of the Mathcad document shown in Figure C3-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C3-2 show the unprocessed data as scatter plots, which are very dense because most of the accesses fell in a small address range and the large size of the data set. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of the upper half of a gaussian distribution. Only half a distribution is generated because of the small size of the input data set. This is more clearly seen in several of the data sets that used a larger input symbol set.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 7$	$mmean = 1$	$mstd = 1.045$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 0$	$xmax = 6$	$xmean = 1$	$xstd = 1.026$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax<xmax, xmax, mmax)$

$top = 7$

$HistPtsA := 0..top$ $BinsA := 0..(top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0..mlast$

Figure C3-1 - Output Data Set ADDRMCUS: Mathcad Document ADDR.MCD

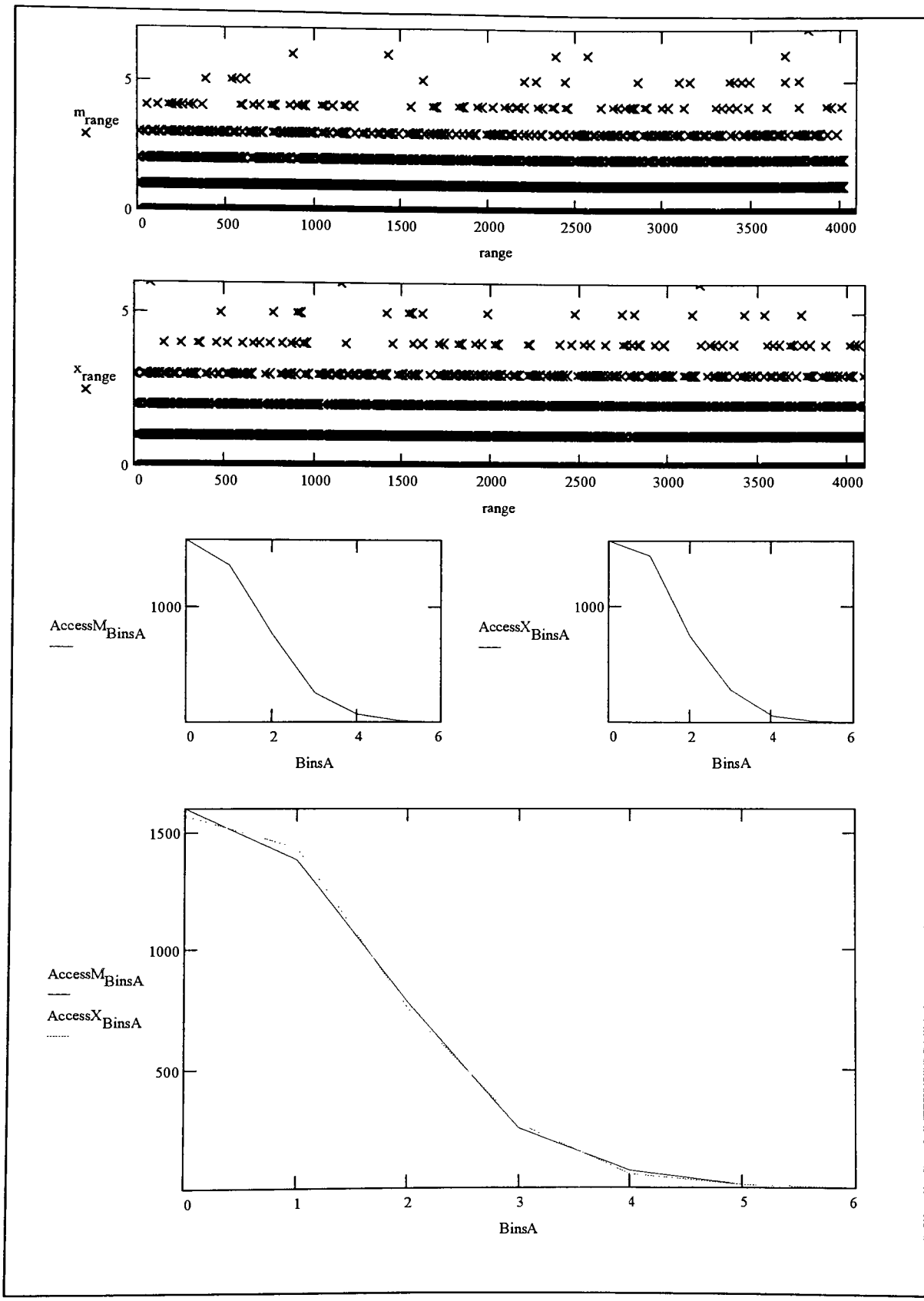


Figure C3-2 - Output Data Set ADDRMCUS: Mathcad Document ADDR.MCD

Appendix C4 Output Data Set ADDRUCGS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the uniformly distributed previous codes (CDEUNIF) and the gaussian distributed symbols (SYMGAUSS). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRUCGS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contain. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdeunif\code.prn symgauss\code.prn addrucgs\mod.prn addrucgs\xor.prn.

The portion of the Mathcad document shown in Figure C4-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C4-2 show the unprocessed data as scatter plots. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of the upper half of a gaussian distribution. Only half a distribution is generated because of the small size of the input data set. This is more clearly seen in several of the data sets that used a larger input symbol set.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := \text{READPRN}(mod)$

Check the array mod parameters.

$mmin := \min(m)$	$mmax := \max(m)$	$mmean := \text{mean}(m)$	$mstd := \text{stdev}(m)$	$mlast := \text{last}(m)$
$mmin = 0$	$mmax = 7$	$mmean = 1$	$mstd = 1.042$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := \text{READPRN}(xor)$

Check the array xor parameters.

$xmin := \min(x)$	$xmax := \max(x)$	$xmean := \text{mean}(x)$	$xstd := \text{stdev}(x)$	$xlast := \text{last}(x)$
$xmin = 0$	$xmax = 6$	$xmean = 1$	$xstd = 1.045$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := \text{if}(mmax < xmax, xmax, mmax)$

$top = 7$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := \text{hist}(HistAryA, m)$

$AccessX := \text{hist}(HistAryA, x)$

$range := 0 .. mlast$

Figure C4-1 Output Data Set ADDRUCGS: Mathcad Document ADDR.MCD

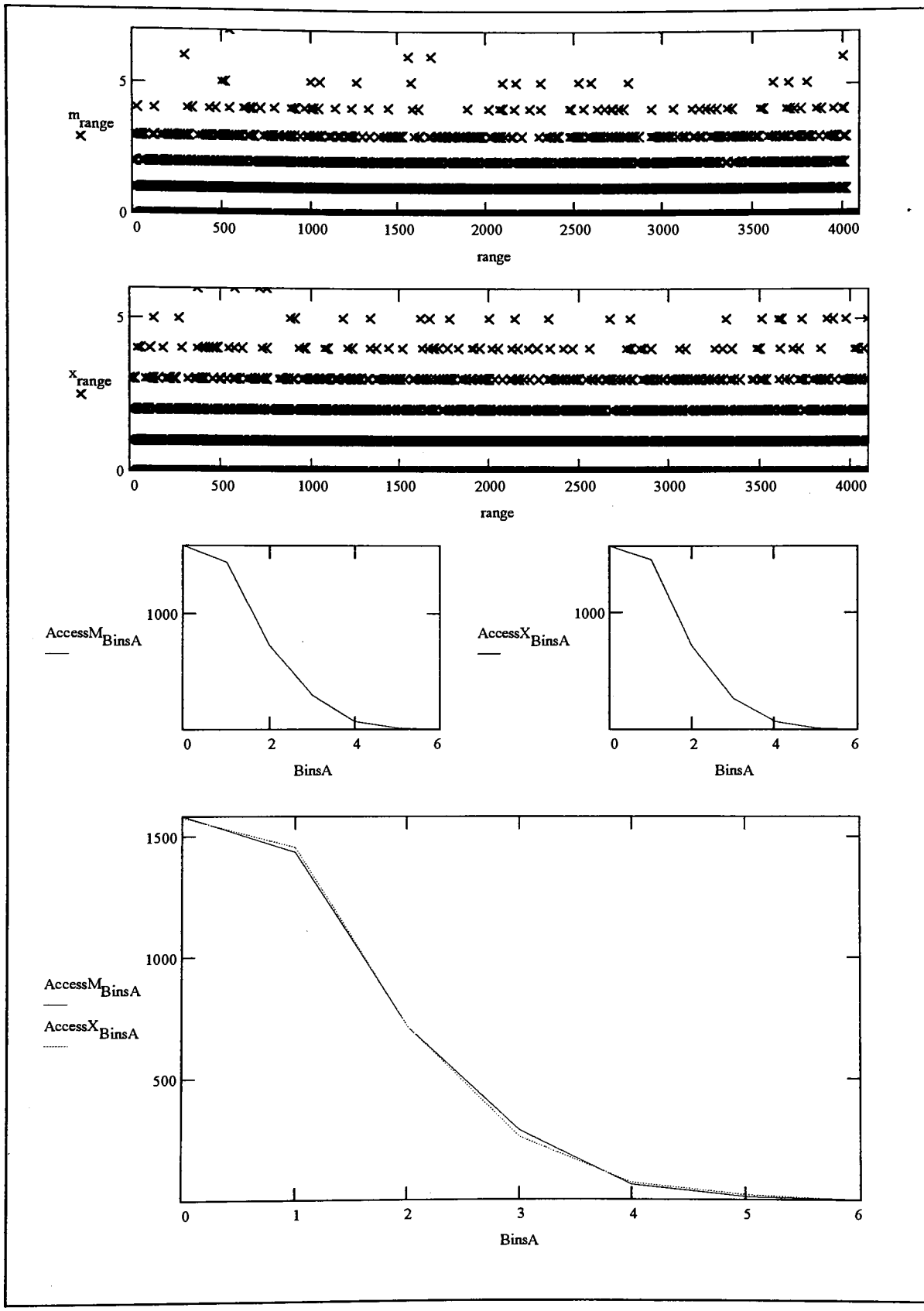


Figure C4-2 - Output Data Set ADDRUCGS: Mathcad Document ADDR.MCD

Appendix C5 Output Data Set ADDRGCGS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the gaussian distributed previous codes (CDEGAUSS) and the gaussian distributed symbols (SYMGAUSS). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRGCGS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contain. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdegauss\code.prn symgauss\code.prn addrgcgs\mod.prn addrgcgs\xor.prn.

The portion of the Mathcad document shown in Figure C5-1 was used to check the data produced by the addr-gen.c. The upper two plots in Figure C5-2 show the unprocessed data as scatter plots, which show that the modulo algorithm produces very different results than the XOR algorithm. This is a direct result of the previous code distribution. Since it is gaussian there are a relatively small number of the possible previous codes used, but the XOR algorithm does a better job of distributing the addresses throughout the memory. The scatter plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. These plots show a highly skewed distribution because of the small size of the input data set (to see the difference between the XOR and the modulo algorithms see appendix 8 which uses a larger input data set).

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := \text{READPRN}(mod)$

Check the array mod parameters.

$mmin := \min(m)$	$mmax := \max(m)$	$mmean := \text{mean}(m)$	$mstd := \text{stdev}(m)$	$mlast := \text{last}(m)$
$mmin = 0$	$mmax = 12$	$mmean = 1$	$mstd = 1.685$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := \text{READPRN}(xor)$

Check the array xor parameters.

$xmin := \min(x)$	$xmax := \max(x)$	$xmean := \text{mean}(x)$	$xstd := \text{stdev}(x)$	$xlast := \text{last}(x)$
$xmin = 0$	$xmax = 8$	$xmean = 1$	$xstd = 1.049$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := \text{if}(mmax < xmax, xmax, mmax)$

$top = 12$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := \text{hist}(HistAryA, m)$

$AccessX := \text{hist}(HistAryA, x)$

$range := 0 .. mlast$

Figure C5-1 Output Data Set ADDRGCGS: Mathcad Document ADDR.MCD

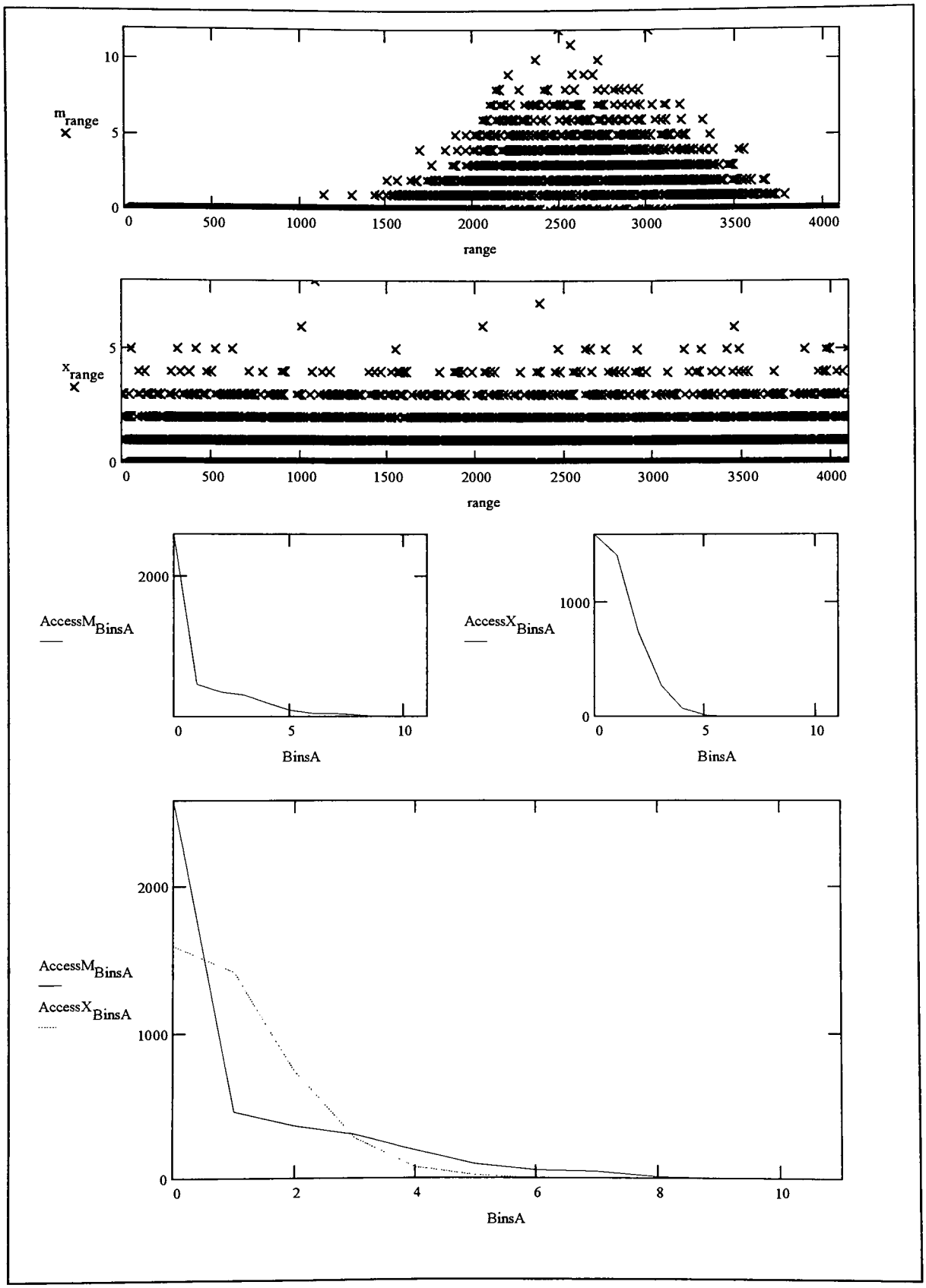


Figure C5-2 Output Data Set ADDRGCGS: Mathcad Document ADDR.MCD

Appendix C6 Output Data Set ADDRMCGS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the my C program distributed previous codes (CDEMYC1) and the gaussian distributed symbols (SYMGAUSS). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRMCGS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdemyc1\code.prn symgauss\code.prn addrmcgs\mod.prn addrmcgs\xor.prn.

The portion of the Mathcad document shown in Figure C6-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C6-2 show the unprocessed data as scatter plots, which are very dense because most of the accesses fell in a small address range and the large size of the data set. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of the upper half of a gaussian distribution. Only half a distribution is generated because of the small size of the input data set. This is more clearly seen in several of the data sets that used a larger input symbol set.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 6$	$mmean = 1$	$mstd = 1.036$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 0$	$xmax = 7$	$xmean = 1$	$xstd = 1.033$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 7$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C6-1 Output Data Set ADDRMCGS: Mathcad Document ADDR.MCD

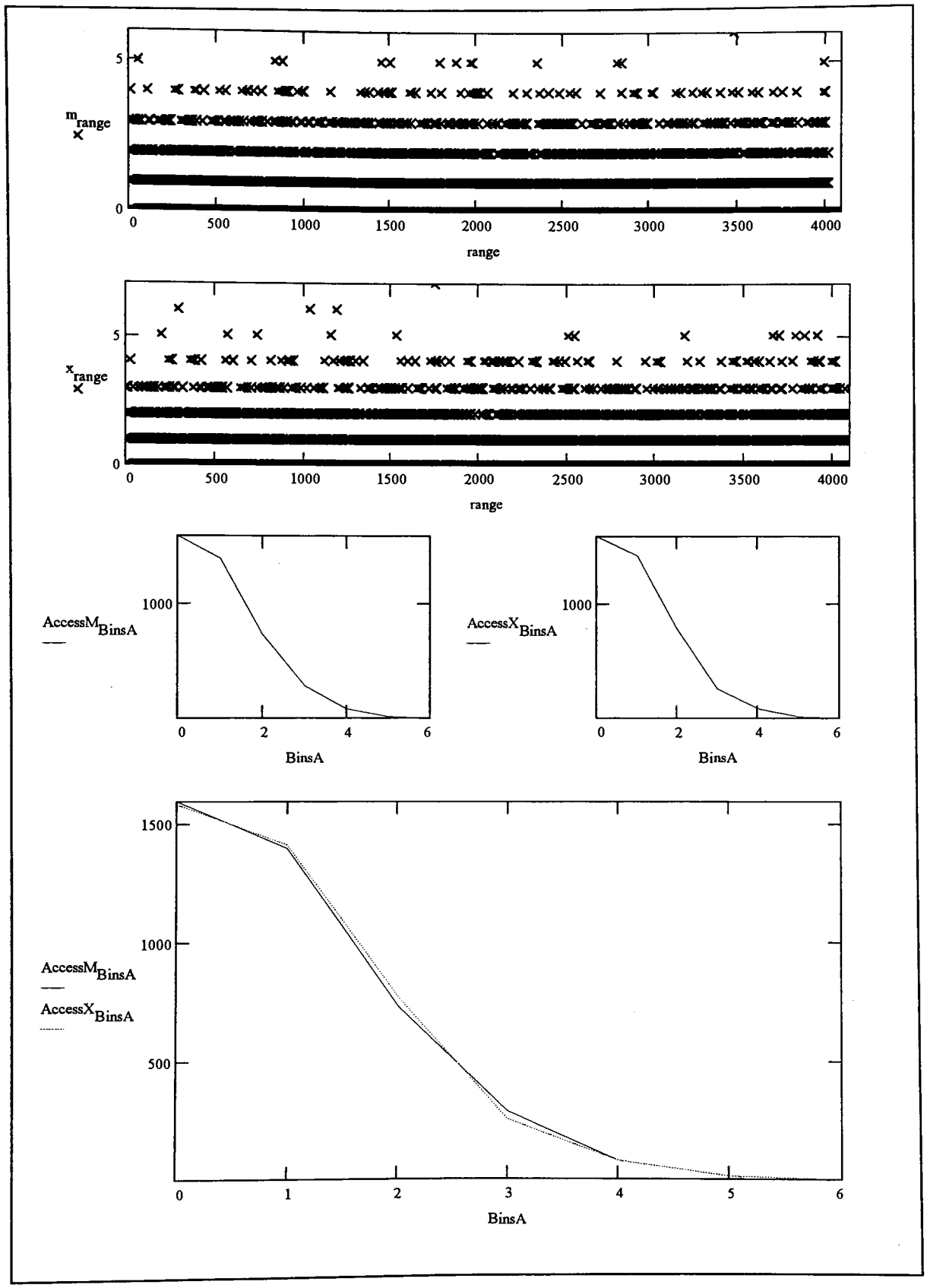


Figure C6-2 - Output Data Set ADDRMCGS: Mathcad Document ADDR.MCD

Appendix C7 Output Data Set ADDRUCMS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the uniformly distributed previous codes (CDEUNIF) and the my C program distributed symbols (SYMMYC1). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRUCMS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdeunif\code.prn symmyc1\code.prn addrucms\mod.prn addrucms\xor.prn.

The portion of the Mathcad document shown in Figure C7-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C7-2 show the unprocessed data as scatter plots, which are very dense due to the large size of the data set. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of a gaussian distribution. As mentioned previously, using a larger data set clearly shows the full distribution shape.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 40$	$mmean = 16$	$mstd = 6.276$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 1$	$xmax = 41$	$xmean = 16$	$xstd = 5.651$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 41$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C7-1 - Output Data Set ADDRUCMS: Mathcad Document ADDR.MCD

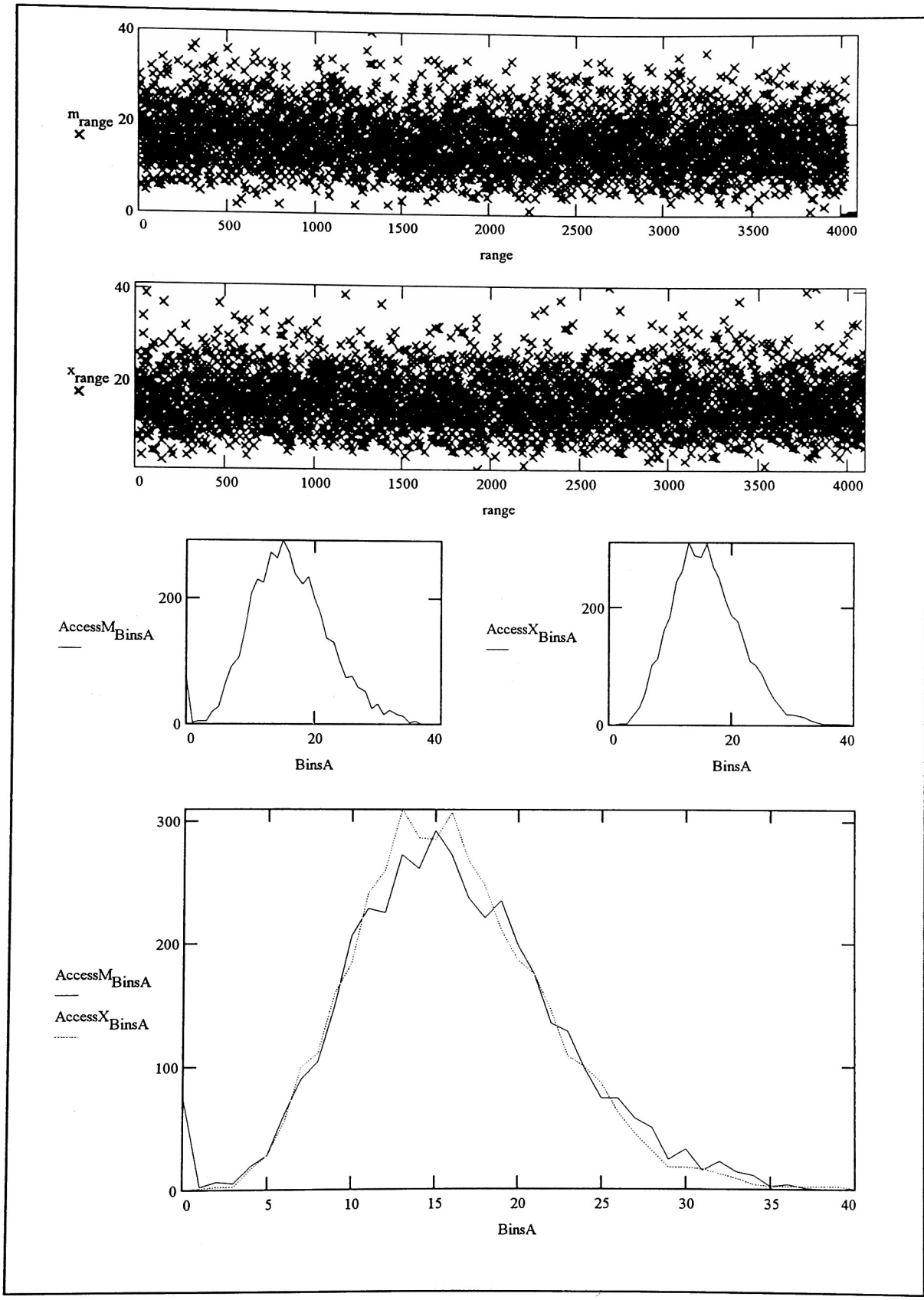


Figure C7-2 - Output Data Set ADDRUCMS: Mathcad Document ADDR.MCD

Appendix C8 Output Data Set ADDRGCMS

This appendix contains the Mathcad document used to check the data that was generated as addresses. This file was produced by the program addr-gen.c. The addresses were produced using the gaussian distributed previous codes (CDEGAUSS) and the my C program distributed symbols (SYMMYC1). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRGCMS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdegauss\code.prn symmyc1\code.prn addrgcms\mod.prn addrgcms\xor.prn.

The portion of the Mathcad document shown in Figure C8-1 was used to check the data produced by the addr-gen.c. The upper two plots in Figure C8-2 show the unprocessed data as scatter plots, which show that the modulo algorithm produces very different results than the XOR algorithm. This is a direct result of the previous code distribution. Since it is gaussian there are a relatively small number of the possible previous codes used, but the XOR algorithm does a better job of distributing the addresses throughout the memory. The scatter plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The modulo plot shows a highly skewed distribution, but, because of the larger input data set size, the XOR exhibits a gaussian shape (compare to appendices C2 and C5). The lowest plot shows both histograms plotted on the same axis.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 99$	$mmean = 16$	$mstd = 21.793$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 1$	$xmax = 43$	$xmean = 16$	$xstd = 6.347$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 99$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C8-1 Output Data Set ADDRGCMS: Mathcad Document ADDR.MCD

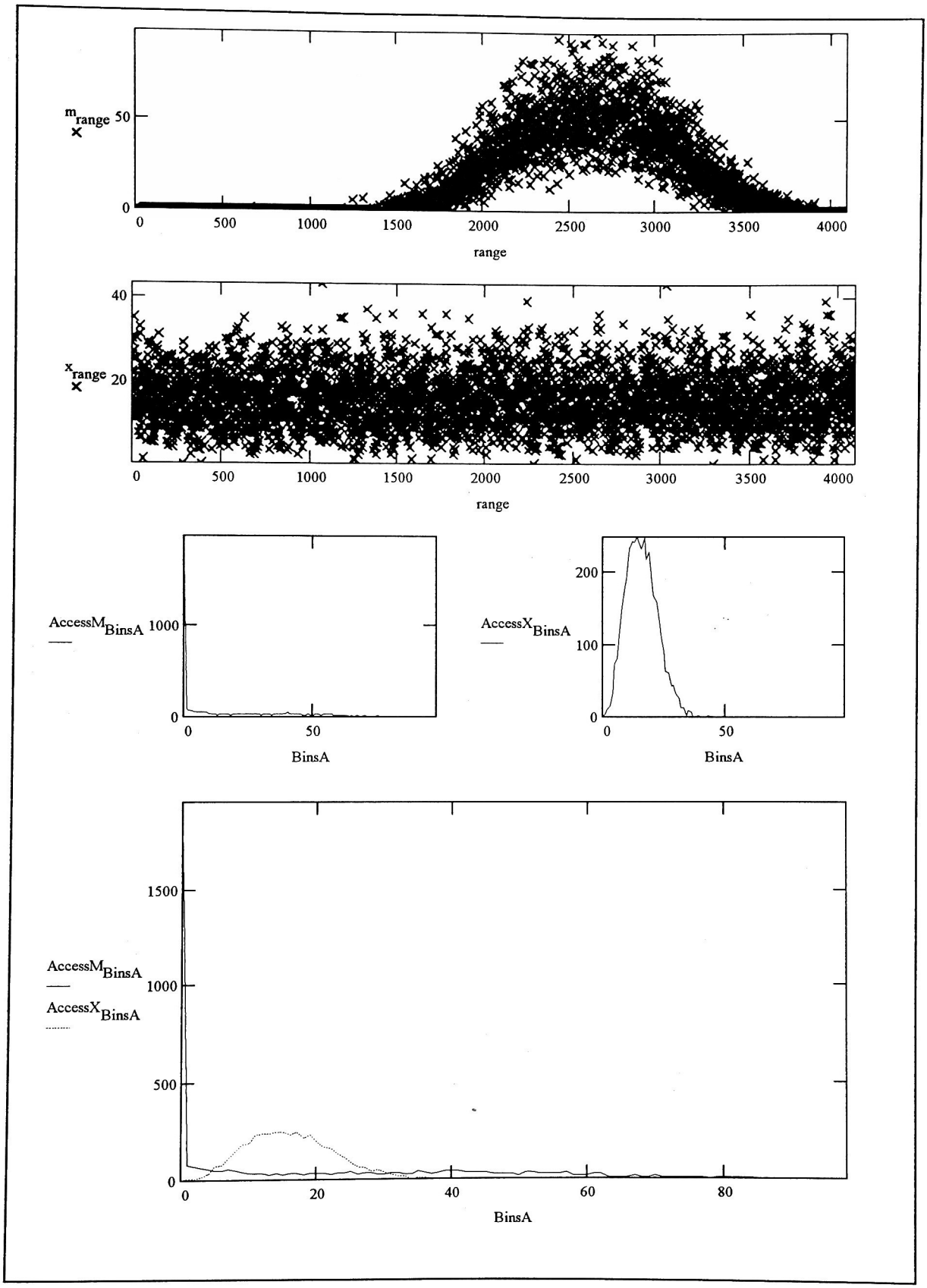


Figure C8-2 - Output Data Set ADDRGCMS: Mathcad Document ADDR.MCD

Appendix C9 Output Data Set ADDRMCMS

This appendix contains the Mathcad documents used to check the data that was generated as addresses, and some additional analysis of this data that will be discussed below. This file was produced by the program addr-gen.c. The addresses were produced using the my C program distributed previous codes (CDEMYC1) and the my C program distributed symbols (SYMMYC1). Due to the file naming constraints of MS-DOS (where the Mathcad document was produced) the data produced was named ADDRMCMS.

Addr-gen.c took the input files and generated mod and XOR addresses from the data they contained. These addresses were saved in files mod.prn and xor.prn. The program used is shown in appendix A4. The command line used was:

addr-gen.exe cdemyc1\code.prn symmyc1\code.prn addrmcms\mod.prn addrmcms\xor.prn.

The portion of the Mathcad document shown in Figure C9-1 was used to check the data produced by the addr-gen.c. It reads the data in, calculates a number of parameters of the data, generates histograms from the data, and plots the data. The upper two plots in Figure C9-2 show the unprocessed data as scatter plots, which are very dense because most of the accesses fell in a small address range and the large size of the data set. The plots show the actual number of times each address was generated. The data in the middle plots shows the same information plotted from the calculated histograms. The lowest plot shows both histograms plotted on the same axis. The plots have the general shape of a gaussian distribution. As mentioned previously, using a larger data set clearly shows the full distribution shape.

This appendix contains two additional Mathcad documents. These will display the results of using a convolution (smooth.c) and some data reduction on the input data. This set of data was chosen because of the large number of values in the input data set for the symbols. This gave a better basis for further analysis. Other than that this set of data is neither better nor worst than some of the other sets, and was chosen at random.

The files mod.prn and xor.prn were first convolved to reduce the extremes of the data. The command lines used were:

\thesis\theodata\smooth.exe .\mod.prn .\mone.prn 31

\thesis\theodata\smooth.exe .\mone.prn .\mtwo.prn 31

\thesis\theodata\smooth.exe .\mtwo.prn .\mthree.prn 31

\thesis\theodata\smooth.exe .\xor.prn .\xone.prn 31

\thesis\theodata\smooth.exe .\xone.prn .\xtwo.prn 31

\thesis\theodata\smooth.exe .\xtwo.prn .\xthree.prn 31.

This data was then analyzed identically in two different documents. One for the mod data, which is called MSMOOTH.MCD and is shown in Figure C9-3, Figure C9-4, and Figure C9-5. And another for the XOR data which is called XSMOOTH.MCD and is shown in Figure C9-6, Figure C9-7, and Figure C9-8.

The set of four plots shown in Figure C9-3 and Figure C9-6 are the plots of the number of times each address was generated for the original data and after each of the three runs through the convolution. As would be expected the mean of the data stays about the same, but the standard deviation is greatly reduced.

Figure C9-4 and Figure C9-7 show the equations used to reduce the number of data points, and the resulting plots. A simple summing and averaging technique was used. The values in 16 consecutive location were added together and then divided by 16. This was done in an attempt to simplify the data in steps to observe if any new information would become visible that had been obscured by the large amount of data. The plots were only produced for the original data set, and the data set after the final convolution. Both plots were displayed as line plots. Finally, Figure C9-5 and Figure C9-8 show the same data displayed as plots using error bars.

None of these plots show any gross unexpected abnormalities. There are some differences between the mod and the XOR results, but in general they are quite comparable. The final plots retain the same general form as the original, but with fewer smoother features because of the operations performed.

Address generation analysis. This is file ADDR.MCD.

Read in the modulo input data.

$m := READPRN(mod)$

Check the array mod parameters.

$mmin := min(m)$	$mmax := max(m)$	$mmean := mean(m)$	$mstd := stdev(m)$	$mlast := last(m)$
$mmin = 0$	$mmax = 41$	$mmean = 16$	$mstd = 6.178$	$mlast = 4.095 \cdot 10^3$

Read in the xor input data.

$x := READPRN(xor)$

Check the array xor parameters.

$xmin := min(x)$	$xmax := max(x)$	$xmean := mean(x)$	$xstd := stdev(x)$	$xlast := last(x)$
$xmin = 1$	$xmax = 38$	$xmean = 16$	$xstd = 5.771$	$xlast = 4.095 \cdot 10^3$

Generate histograms for the input data.

$top := if(mmax < xmax, xmax, mmax)$

$top = 41$

$HistPtsA := 0 .. top$ $BinsA := 0 .. (top - 1)$

$HistAryA_{HistPtsA} := HistPtsA$ This turns a range into a vector for the histogram function.

$AccessM := hist(HistAryA, m)$

$AccessX := hist(HistAryA, x)$

$range := 0 .. mlast$

Figure C9-1 Output Data Set ADDRMCMS: Mathcad Document ADDR.MCD

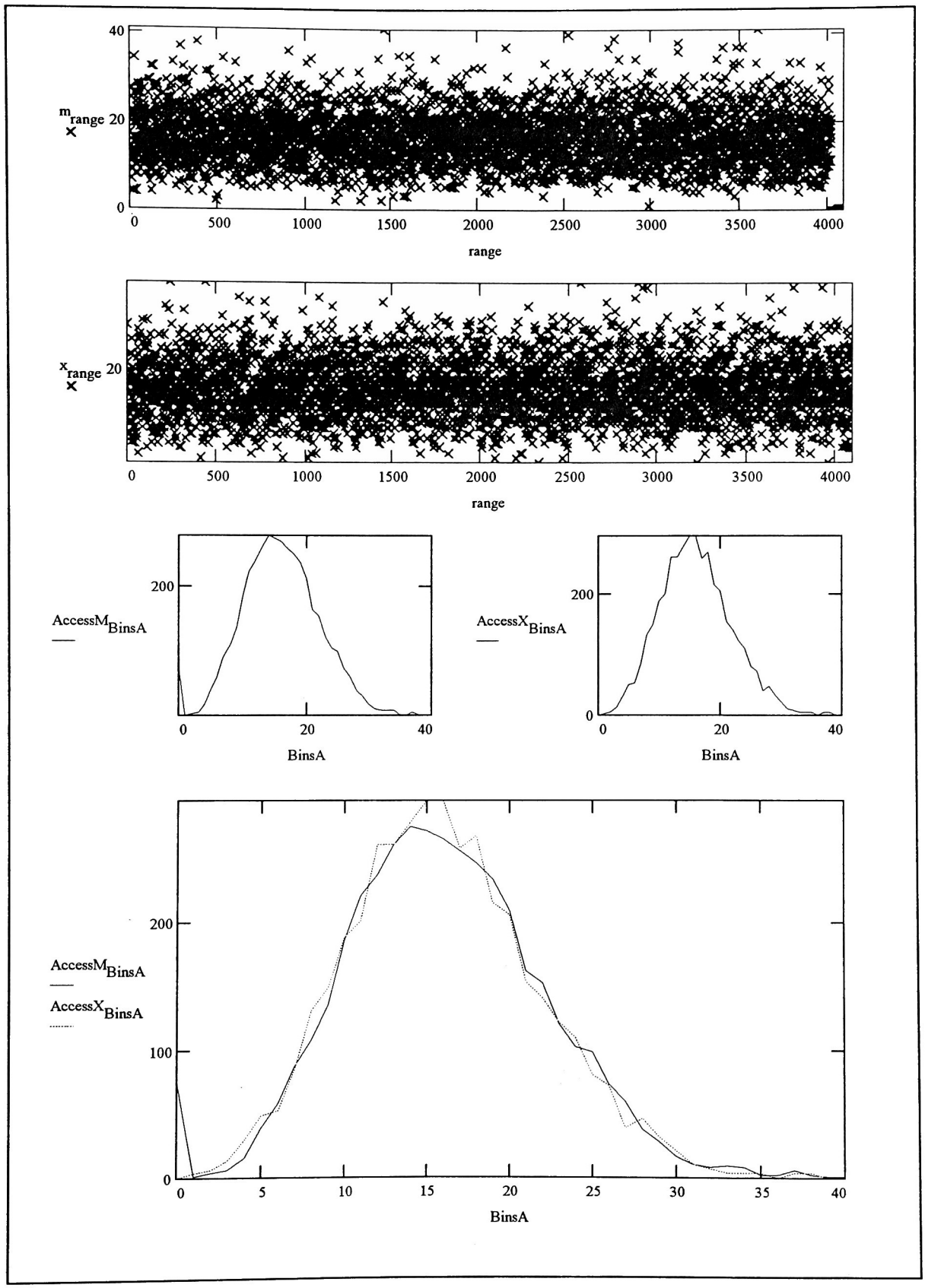


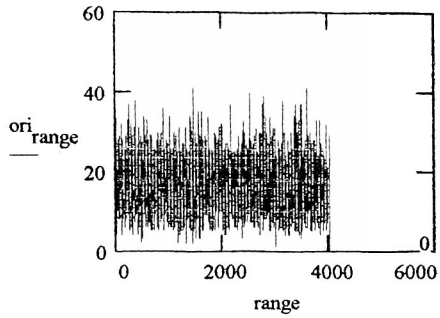
Figure C9-2 - Output Data Set ADDRMCMS: Mathcad Document ADDR.MCD

File MSMOOTH.MCD. Results of smoothing, and then reducing the number of data points using matrix manipulations.

ori := READPRN(mod) one := READPRN(mone)

two := READPRN(mtwo) thr := READPRN(mthree)

range := 0..4095

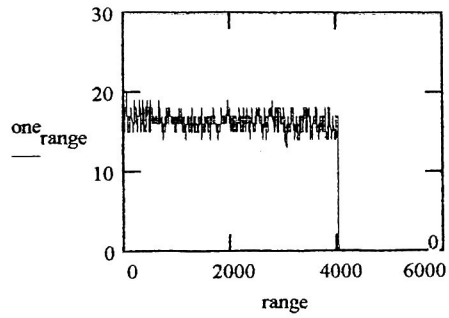


orimean := mean(ori) oristd := stdev(ori)

orimean = 16 oristd = 6.178

orimax := max(ori) orimin := min(ori)

orimax = 41 orimin = 0

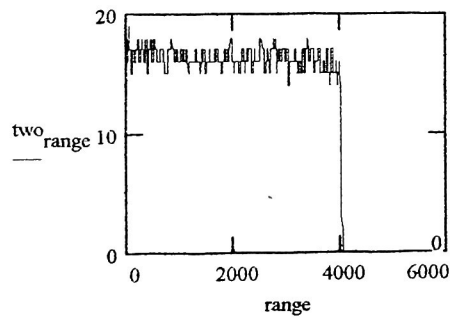


onemean := mean(one) onestd := stdev(one)

onemean = 16.006 onestd = 2.335

onemax := max(one) onemin := min(one)

onemax = 21 onemin = 0

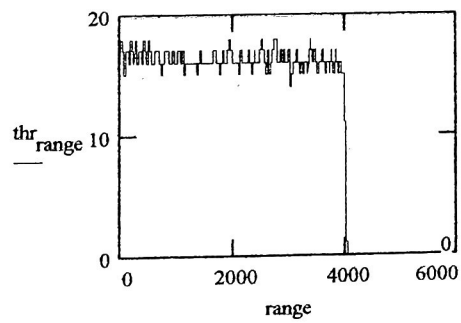


twomean := mean(two) twostd := stdev(two)

twomean = 16.001 twostd = 2.231

twomax := max(two) twomin := min(two)

twomax = 20 twomin = 0



thrmean := mean(thr) thrstd := stdev(thr)

thrmean = 15.999 thrstd = 2.191

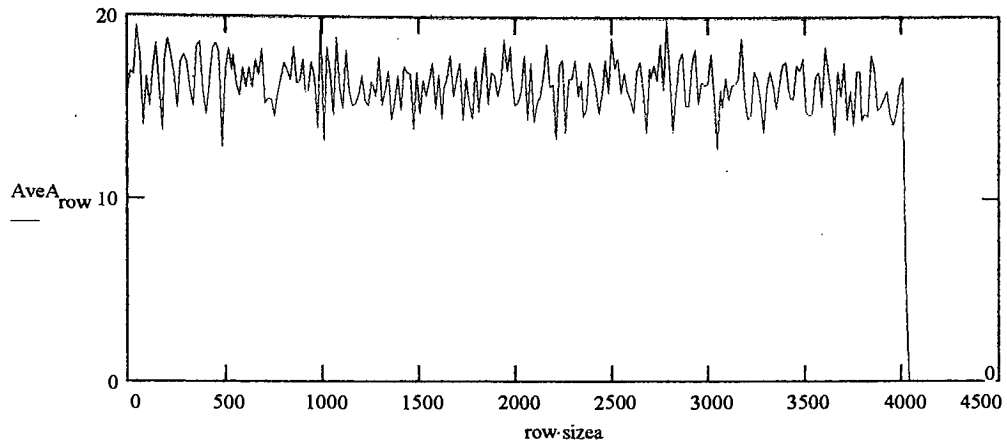
thrmax := max(thr) thrmin := min(thr)

thrmax = 19 thrmin = 0

Figure C9-3 - Data set ADDRMCMS: Mathcad Document MSMOOTH.MCD

sizea := 16 ColEnd := (sizea - 1) RowEnd := $\left(\frac{4096}{sizea} - 1\right)$ col := 0 .. ColEnd row := 0 .. RowEnd

InΛ := ori $SqΛ_{row,col} := InΛ_{sizea \cdot row + col}$ $ΛveΛ_{row} := \frac{1}{sizea} \cdot \sum_{col} SqΛ_{row,col}$



sizeb := 16 ColEndb := (sizeb - 1) RowEndb := $\left(\frac{4096}{sizeb} - 1\right)$ colb := 0 .. ColEndb rowb := 0 .. RowEndb

InB := thr $SqB_{rowb,colb} := InB_{sizeb \cdot rowb + colb}$ $ΛveB_{rowb} := \frac{1}{sizeb} \cdot \sum_{colb} SqB_{rowb,colb}$

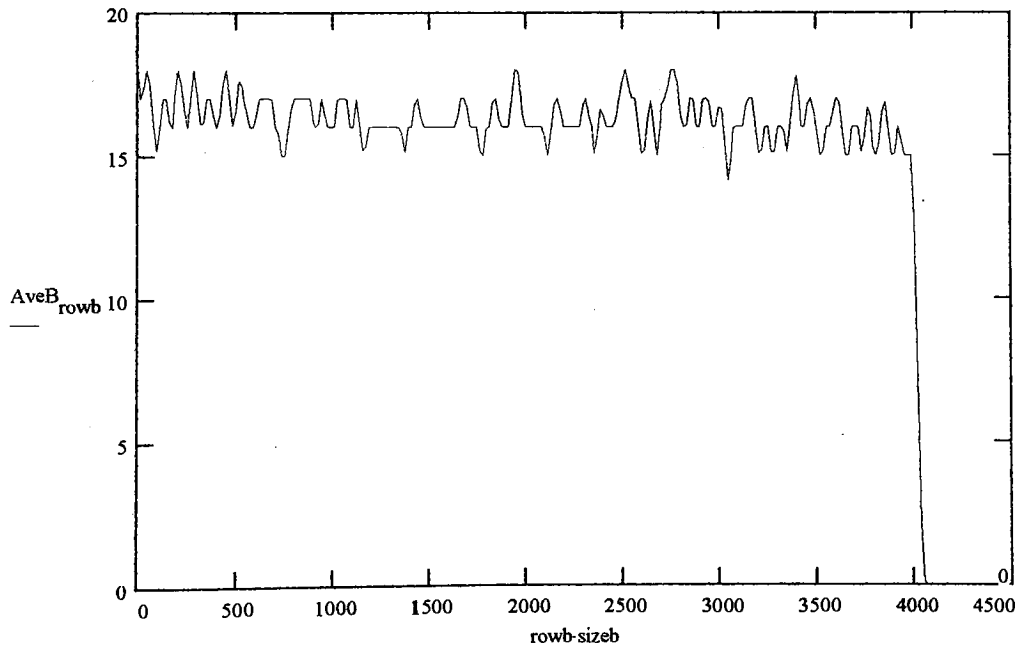


Figure C9-4 - Data set ADDRMCMS: Mathcad Document MSMOOTH.MCD

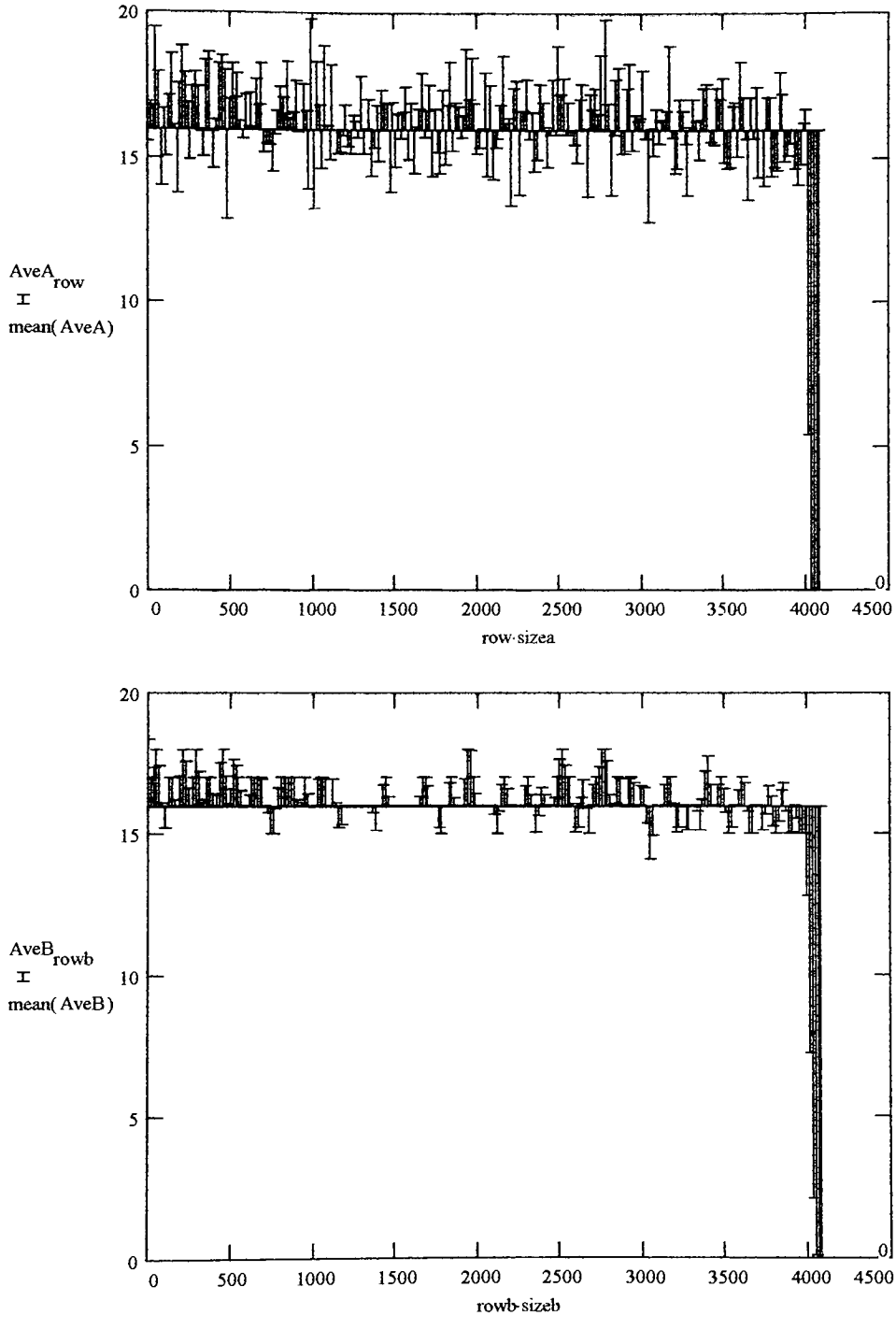


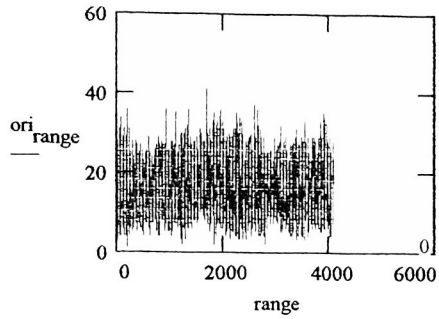
Figure C9-5 - Data set ADDRMCMS: Mathcad Document MSMOOTH.MCD

File XSMOOTH.MCD. Results of smoothing, and then reducing the number of data points using matrix manipulations.

ori := READPRN(xor) one := READPRN(xone)

two := READPRN(xtwo) thr := READPRN(xthree)

range := 0 .. 4095

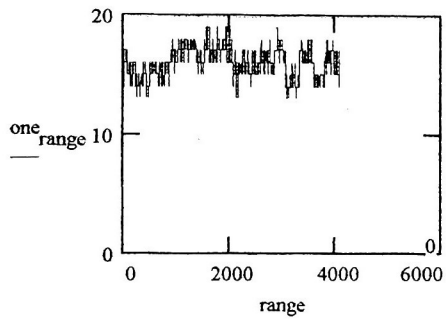


orimean := mean(ori) oristd := stdev(ori)

orimean = 16 oristd = 5.868

orimax := max(ori) orimin := min(ori)

orimax = 41 orimin = 1

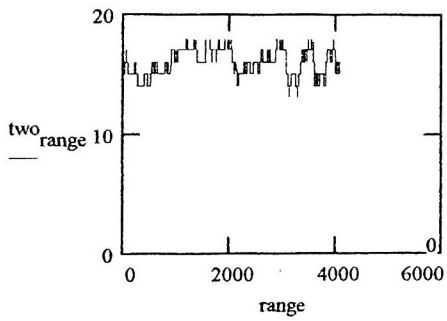


onemean := mean(one) onestd := stdev(one)

onemean = 15.99 onestd = 1.228

onemax := max(one) onemin := min(one)

onemax = 19 onemin = 13

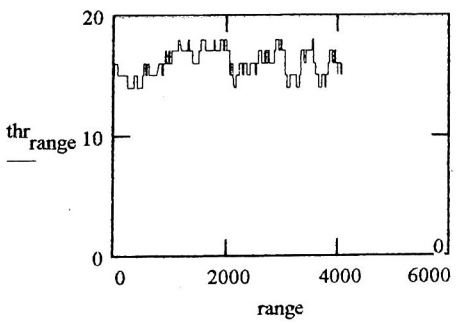


twomean := mean(two) twostd := stdev(two)

twomean = 15.986 twostd = 1.154

twomax := max(two) twomin := min(two)

twomax = 18 twomin = 13



thrmean := mean(thr) thrstd := stdev(thr)

thrmean = 15.988 thrstd = 1.131

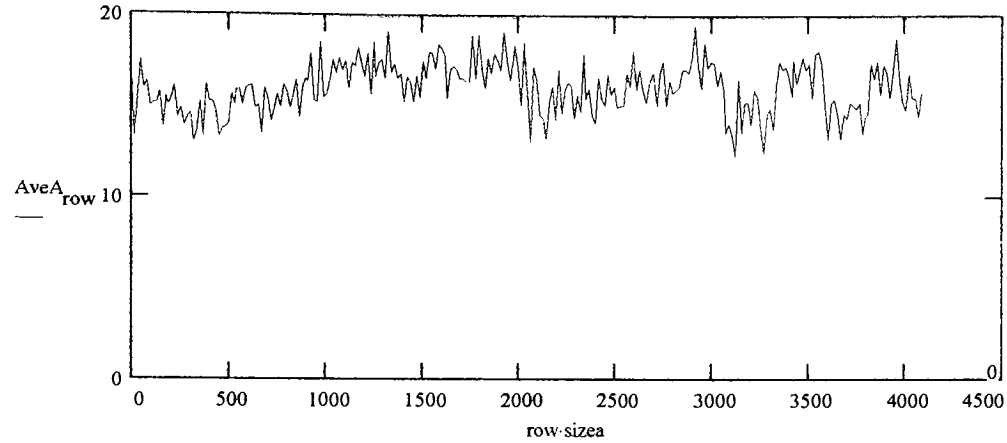
thrmax := max(thr) thrmin := min(thr)

thrmax = 18 thrmin = 14

Figure C9-6 - Data set ADDRMCMS: Mathcad Document XSMOOTH.MCD

$sizea := 16$ $ColEnd := (sizea - 1)$ $RowEnd := \left(\frac{4096}{sizea} - 1\right)$ $col := 0..ColEnd$ $row := 0..RowEnd$

$InA := ori$ $SqA_{row,col} := InA_{sizea \cdot row + col}$ $AveA_{row} := \frac{1}{sizea} \cdot \sum_{col} SqA_{row,col}$



$sizeb := 16$ $ColEndb := (sizeb - 1)$ $RowEndb := \left(\frac{4096}{sizeb} - 1\right)$ $colb := 0..ColEndb$ $rowb := 0..RowEndb$

$InB := thr$ $SqB_{rowb,colb} := InB_{sizeb \cdot rowb + colb}$ $AveB_{rowb} := \frac{1}{sizeb} \cdot \sum_{colb} SqB_{rowb,colb}$

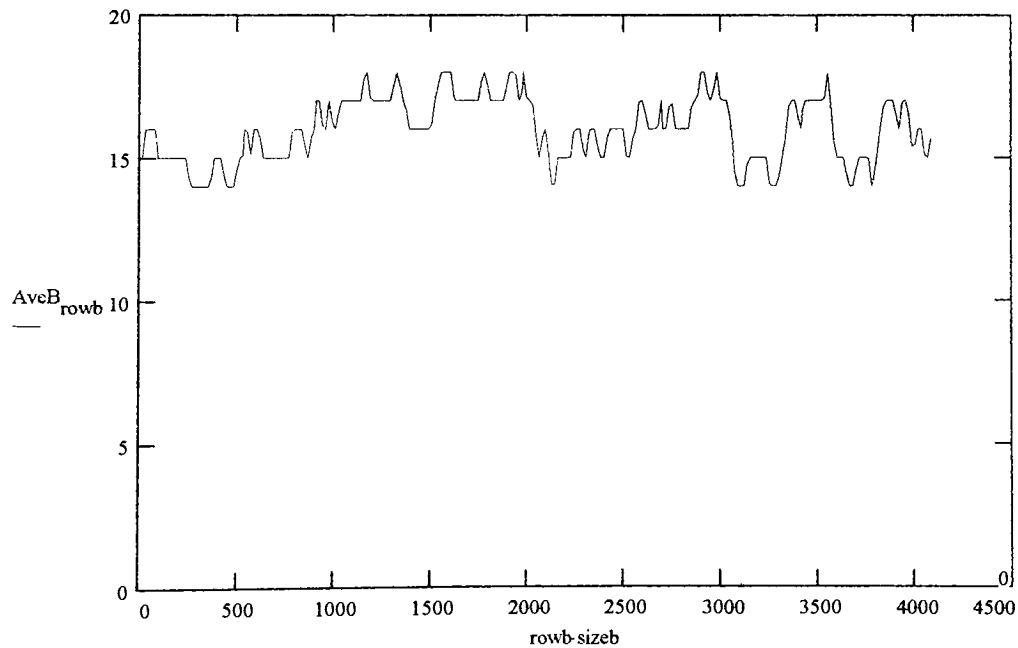
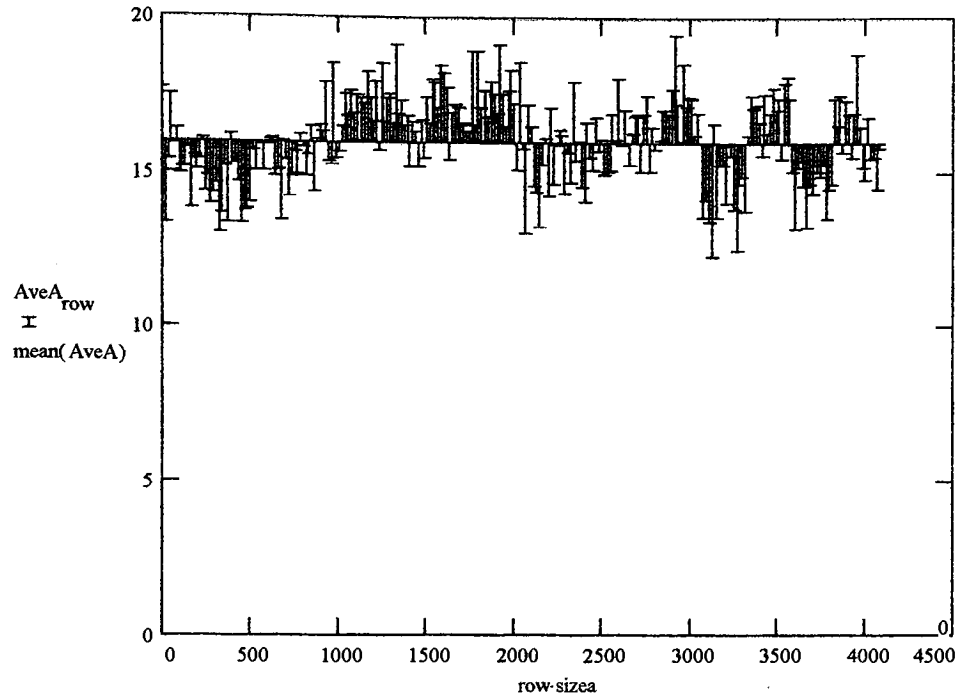


Figure C9-7 - Data set ADDRMCMS: Mathcad Document XSMOOTH.MCD



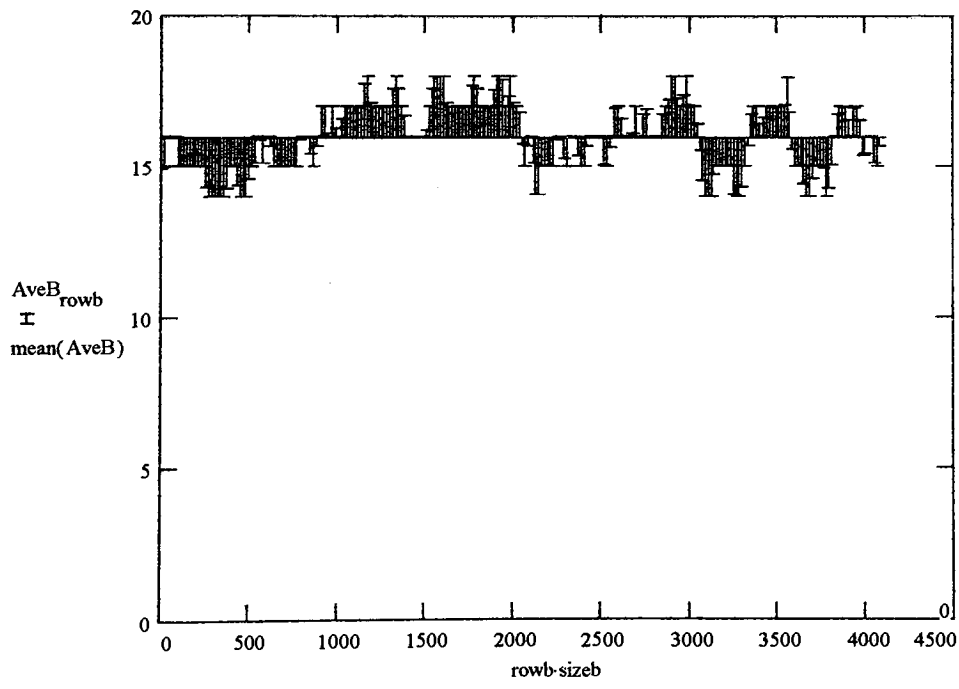


Figure C9-8 - Data set ADDRMCMS: Mathcad Document XSMOOTH.MCD

APPENDIX D - VHDL Source Code Files

Appendix D1 - Source Code for HashTB.vhdl

This appendix contains the source code listing for the VHDL file HashTB.vhdl. The purpose of this program was to provide the stimulus to the device under test and to check the results. The stimulus was values read out of a file opened by the testbench, the results were histogrammed and then compared to the values read from another file that was opened by the testbench. The histogram values were stored in a third file that was opened by the testbench.

Supporting information is in HashPKG.vhdl. This contained the definitions of the types, subtypes, and subprograms needed by the testbench. The package textio is also referenced because the files that are used are opened as files of type text.

INPUTS: file of data representing symbols,
file of results produced by the C code version of the algorithm.

OUTPUTS: file representing the histogram of the addresses produced.

```
-------------------------------------------------------------------------------
--  HashTB.vhdl
--  This file is to read data from files, feed it to the hashing algorithm,
--  handle the output from the aformentioned, and compare the generated
--  results with known good results from the C code.
-------------------------------------------------------------------------------

USE std.textio.all;
USE work.HashPKG.all;

-------------------------------------------------------------------------------
--  entity for the top level (the Test Bench)
-------------------------------------------------------------------------------
ENTITY HashTB IS
  --  The top level test bench has no ports, therefore an empty entity
END HashTB;


-------------------------------------------------------------------------------
--  architecture for top level entity
-------------------------------------------------------------------------------
ARCHITECTURE HashTBarch OF HashTB IS
  SIGNAL        InData          : symbolST := symbolST'right;
  SIGNAL        OutData         : symbolST;
  SIGNAL        Clock           : bit;
  SIGNAL        ReadEnable      : enableT := stop;
  SIGNAL        WriteEnable     : enableT := stop;
  SIGNAL        BeginOperation  : boolean := false;
  SIGNAL        size            : integer;
  SIGNAL        EndOfInput      : boolean := false;
  SIGNAL        RunClock        : boolean := true;
  SIGNAL        CountArray      : AddrArrayT(TwelveBitST);
BEGIN  --  HashTBarch

  -----------------------------------------------------------------------------
  --  Define a concurrent clock just to keep things running.
  --  NOTE:  the clock is turned off after the work is done.  This allows the
  --  simulator to run with the -until complete switch which means that no
  --  time value is needed.
  -----------------------------------------------------------------------------
  Clock  <= NOT Clock after HalfPeriod WHEN RunClock else
            Clock;

  cBeginOperation: BeginOperation <=
              true after (HalfPeriod + HalfPeriod/2) WHEN (NOT EndOfInput) else
              false WHEN (EndOfInput) else
              true;         --  impromptu startup delay

   -----------------------------------------------------------------------------
   --  Process:   TimeOutClock
   --  Purpose:   To stop clock after all the input has been read, but giving
   --             enough extra clocks to finish any data in the pipeline.
   --             This process exists purely to assist the simulator.
   -----------------------------------------------------------------------------
   TimeOutClock : PROCESS (Clock)
     VARIABLE   count           : natural := 0;
   BEGIN  --  PROCESS TimeOutClock
     IF (Clock = '1' and EndOfInput) THEN
       IF (count < 10) THEN
         count := count + 1;
       ELSE
         RunClock <= False;
       END IF;
     END IF;
   END PROCESS TimeOutClock;
```

```
-------------------------------------------------------------------------------
--  Process:    ReadFile
--  Purpose:    Read lines (each assumed to contain only one integer with a
--              value between 0 and 255) from an input file and put them unto
--              a signal to make them usable by whatever else wants them.
-------------------------------------------------------------------------------
ReadFile : PROCESS (Clock)
  VARIABLE    line_buffer             : line;
  file        InputFile               : TEXT is in "InputFile";
    --  override the default file declaration in text I/O package
  VARIABLE    convert                 : symbolST;
BEGIN  --  PROCESS ReadFile
  IF (Clock'event AND Clock = '0' AND BeginOperation) THEN
    readline (InputFile, line_buffer);
        ---------------------------------------------------------------------
        --  This construct will cause a runtime error if the input file does
        --  not have a blank line as the last line when 'length is used to
        --  detect the end of file condition.
        ---------------------------------------------------------------------
    size <= line_buffer'length;         --  used for Resdisp (debugging)
    IF (line_buffer'length = 0) THEN
      ASSERT false
        REPORT "End of input file"
          SEVERITY note;
      EndOfInput <= true;
      ReadEnable <= stop after (HalfPeriod/2);
    ELSE
      read (line_buffer, convert);
      InData <= convert after (HalfPeriod/2);
      ReadEnable <= run after (HalfPeriod/2);
    END IF;
  END IF;
END PROCESS ReadFile;

-------------------------------------------------------------------------------
--  Process:    WriteFile
--  Purpose:    Take data from the test bench and write it out to a file.  It
--              is assumed that only one integer value will be written out to
--              each line in the output file.
-------------------------------------------------------------------------------
WriteFile : PROCESS (Clock)
  VARIABLE    line_buffer             : line;
  file        OutputFile              : text is out "OutputFile";
BEGIN  --  PROCESS WriteFile
  IF ( Clock'event AND Clock = '0' AND WriteEnable = run) THEN
    CountArray(OutData) <= CountArray(OutData) + 1;
    write (line_buffer, OutData);
    writeline (OutputFile, line_buffer);
  END IF;
END PROCESS WriteFile;
```

```
  ------------------------------------------------------------------------------
  --  Process:    CompareArray
  --  Purpose:    When all the input data has been used this process runs.  It
  --              reads the known good data, compares the data produced here,
  --              and it writes this data out.
  ------------------------------------------------------------------------------
  CompareArray : PROCESS (WriteEnable)
    FILE        CheckFile          : TEXT is in "CheckFile";
    VARIABLE    CheckArray         : AddrArrayT(TwelveBitST);
    VARIABLE    Init               : boolean := TRUE;
    VARIABLE    line_buffer        : line;
    FILE        OutputArrayFile    : text is out "OutputArrayFile";
  BEGIN  --  PROCESS CompareArray
    IF (Init) THEN
      FillArray(CheckArray, CheckFile);
      Init := false;
    ELSIF (WriteEnable'event AND WriteEnable = STOP) THEN
      CompareCounts : FOR Step IN 0 to (CountArray'length - 1) LOOP
        ASSERT CheckArray(Step) = CountArray(Step)
          REPORT "Arrays don't match."
            SEVERITY error;
        write (line_buffer, CountArray(Step));
        writeline (OutputArrayFile, line_buffer);
      END LOOP CompareCounts;
      ASSERT false
        REPORT "Comparison is complete."
          SEVERITY note;
    END IF;
  END PROCESS CompareArray;

  ------------------------------------------------------------------------------
  --  instantiate the component to test
  --  (NOTE: the declaration is in the package HashPKG.vhdl)
  ------------------------------------------------------------------------------
  UXBEH : HashAlgXbehv
    port map (EnableIn => ReadEnable,
              EnableOut => WriteEnable,
              SymbolIn => InData,
              AddrOut => OutData,
              Clock => Clock);

END HashTBarch;


------------------------------------------------------------------------------
--  configuration section
------------------------------------------------------------------------------
CONFIGURATION TBconfigBehv of HashTB is
  FOR HashTBarch
    FOR UXBEH : HashAlgXbehv
      USE ENTITY work.HashAlgXbehv(HashAlgXbehvArch);
    END FOR;
  END FOR;
END TBconfigBehv;
```

```
CONFIGURATION TBconfigStru of HashTB is
  FOR HashTBarch
    FOR UXBEH : HashAlgXbehv USE ENTITY work.HashConvert(HashConvertArch);
      FOR HashConvertArch
        FOR UXSTR : HashAlgXstru USE ENTITY work.HashAlgXstru(HashAlgXstruArch);
          FOR HashAlgXstruArch
            FOR ALL : Reg1 USE ENTITY work.Reg1(Reg1Arch);
            END FOR;
            FOR ALL : AndGate USE ENTITY work.AndGate(AndGateArch);
            END FOR;
            FOR UNCLK : InvGate USE ENTITY work.InvGate(InvGateArch);
            END FOR;
            FOR UMEM : HashMem USE ENTITY work.HashMem(HashMemArch);
            END FOR;
            FOR UMUX : Mux12 USE ENTITY work.Mux12(Mux12Arch);
            END FOR;
            FOR ALL : Reg12 USE ENTITY work.Reg12(Reg12Arch);
            END FOR;
            FOR ALL : XorMulti USE ENTITY work.XorMulti(XorMultiArch);
            END FOR;
          END FOR;
        END FOR;
      END FOR;
    END FOR;
  END FOR;
END TBconfigStru;
```

Appendix D2 - Source Code for HashAlgXbehv.vhdl

This appendix contains the source code listing for the VHDL file HashAlgXbehv.vhdl. The purpose of this program was to model the XOR hashing algorithm as a behavorial VHDL model. All of the stimulus was provided by the testbench, and all of the results were returned to the testbench.

Supporting information is in HashPKG.vhdl. This contained the definitions of the types, subtypes, and subprograms needed by the algorithm. To do the calculations, a data file representing the previous codes was needed. This file was opened and read by the algorithm, but no changes were made to any of the data, it was only read. The package textio is also referenced because the file that was used was opened as a file of type text.

INPUTS: file of data representing the previous codes.

```
--------------------------------------------------------------------------------
--  HashAlgXbehv.vhdl
--  This file will hold the vhdl to implement the behavioral version of the
--  XOR hashing algorithm.  It will accept data from the test bench, generate
--  the address using an internal array (which it will have to fill), and send
--  the address back to the test bench.
--------------------------------------------------------------------------------

USE work.HashPkg.all;
USE std.textio.all;

--------------------------------------------------------------------------------
--  entity for the behavioral version of the XOR hashing algorithm
--------------------------------------------------------------------------------
ENTITY HashAlgXbehv IS
  port( EnableIn  : in EnableT;
        EnableOut : buffer EnableT;
        SymbolIn  : in SymbolST;
        AddrOut   : buffer TwelveBitST;
        Clock     : in bit);
  BEGIN
    ASSERT false
      REPORT "Executing the behavorial version of the algorithm."
        SEVERITY note;
END HashAlgXbehv;

--------------------------------------------------------------------------------
--  architecture for the behavioral version of the XOR hashing algorithm
--------------------------------------------------------------------------------
ARCHITECTURE HashAlgXbehvArch OF HashAlgXbehv IS
  SIGNAL       PcodeVal         : TwelveBitST := TwelveBitST'right;
  SIGNAL       MemVal           : TwelveBitST;
  SIGNAL       First            : boolean := true;
  FILE         PcodeFile        : TEXT is in "PcodeFile";
BEGIN  --  HashAlgXbehvArch

  --------------------------------------------------------------------------------
  --  Process:    NextCode
  --  Purpose:    This process latches in the data from the memory that will
  --              become the next previous code.  It also has to prime the
  --              value using the first symbol read.
  --------------------------------------------------------------------------------
  NextCode : PROCESS (Clock)
    --  unused declaration area
  BEGIN  --  PROCESS NextCode
    IF (Clock'event and Clock  = '0') THEN
      IF (EnableIn = RUN) THEN
        IF (First) THEN
          PcodeVal <= SymbolIn;
          First <= false;
        ELSE
          PcodeVal <= MemVal;
        END IF;
      END IF;
    END IF;
  END PROCESS NextCode;
```

```
  ------------------------------------------------------------------------
  --  Process:    Enable
  --  Purpose:    This process pipelines the enable signal in parallel with the
  --              data.
  ------------------------------------------------------------------------
  Enable : PROCESS (Clock)
    --  unused declaration area
  BEGIN  --  PROCESS Enable
    IF (Clock'event and Clock = '1') THEN
      IF (EnableIn = run AND NOT First) THEN --delay until we start reading data
        EnableOut <= run;
      ELSIF (EnableIn = stop) THEN
        EnableOut <= stop;
      END IF;
    END IF;
  END PROCESS Enable;

  ------------------------------------------------------------------------
  --  Process:    XORhash
  --  Purpose:    This is the process that actually does the hashing operations
  --              on the data.
  ------------------------------------------------------------------------
  XORhash : PROCESS (Clock)
    VARIABLE      Harg              : natural;
  BEGIN  --  PROCESS XORhash
    IF (Clock'event and Clock = '1') THEN
      --  xor the values, shift up, and add -> 8 bits + 12 bits = 20 bits
      Harg := ( (IntXor(SymbolIn, Reverse8bitArray(SymbolIn), 8)) * (2**12) )
                + PcodeVal;
      --  need to shift down by 8 bits -> 20 - 8 = 12
      AddrOut <= IntXor(ShiftDown(Harg, 20, 8), PcodeVal, 12);
    END IF;
  END PROCESS XORhash;

  ------------------------------------------------------------------------
  --  Process:    Memory
  --  Purpose:    This process will perform the task of a memory device,
  --              providing the next previous code whenever the address
  --              changes.
  ------------------------------------------------------------------------
  Memory : PROCESS (AddrOut)
    VARIABLE      PcodeArray        : AddrArrayT(TwelveBitST);
    VARIABLE      Start             : boolean := true;
  BEGIN  --  PROCESS Memory
    IF ( Start ) THEN
      FillArray(PcodeArray, PcodeFile);
      Start := false;
      MemVal <= PcodeArray(AddrOut) after (HalfPeriod/2);
    ELSE
      MemVal <= PcodeArray(AddrOut) after (HalfPeriod/2);
    END IF;                                  --  Address
  END PROCESS Memory;

END HashAlgXbehvArch;
```

Appendix D3 - Source Code for HashPKG.vhdl

This appendix contains the source code listing for the VHDL file HashPKG.vhdl. The purpose of this package was to provide support code for any of the other design units. Contained in this file were all of the types, subtypes, and constants needed. It also contained most of the subprograms (functions and procedures), the rest are in dataconv_pkg.vhdl.

This package was written specifically for this algorithm, and no attempt was made to generalize its contents. Even with that disclaimer, it can be seen that most of the information contained in the package is of a general nature that holds the possibility of application to other designs.

The package textio is referenced because the files that are used are opened as files of type text. The package dataconv is referenced to provide the conversion functions to and from bit_vectors and integers.

```
USE std.textio.all;
USE work.dataconv_pkg.all;

-------------------------------------------------------------------------------
--  define a package for types etc.
-------------------------------------------------------------------------------
PACKAGE HashPKG IS

  SUBTYPE SymbolST IS natural range 0 to 255;

  TYPE EnableT IS (stop, run);

  SUBTYPE TwelveBitST IS natural range 0 to 4095;

  TYPE AddrArrayT IS array(natural range <>) of TwelveBitST;

  CONSTANT      HalfPeriod       : time := 50 ns;

  PROCEDURE FillArray (VARIABLE ArrayName : out AddrArrayT;
                       VARIABLE FileName : in TEXT);
    -- This procedure reads (textio) from a file and puts the information
    -- into an array which it passes out.

  FUNCTION ShiftDown (value, size, count : in natural) RETURN natural;
    -- This function takes in a non-negative integer, the number of bits
    -- needed to represent this value, and how many bits to shift the value
    -- by.  It converts the value to a bit vector and then shifts it "down",
    -- i.e. it returns the high order bits.

  FUNCTION IntXor (int1, int2, int1_2_len: in integer) RETURN integer;
    -- This function takes in two integers (int1 and int2), it also needs the
    -- length of the bit string necessary to hold the largest value possible in
    -- the input integers (int1_2_len).  It takes the two integers and converts
    -- them into bit_vectors, does a bit-wise XOR, and then converts the
    -- resulting bit_vector back into an integer and returns it.

  FUNCTION TBtoBitConv (TBval : in TwelveBitST) RETURN bit_vector;
    -- This is a port conversion function to change TwelveBitST (subtype of
    -- natural) into a 12 bit bit vector.  The width is determined by the
    -- range of the subtype.

  FUNCTION BitToTBConv (BVval : in bit_vector) RETURN TwelveBitST;
    -- This is a port conversion function to change a 12 bit bit vector into
    -- TwelveBitST (subtype of natural).  The width is determined by the
    -- range of the subtype.
```

```
-- Begining of specifications for 8bit reverse bits array
TYPE Array8bitT IS array (natural range <>) of SymbolST;
  -- this is a package declaration, the array delcaration must be a CONSTANT
constant Reverse8BitArray : Array8bitT(SymbolST) := (
       0, 128,  64, 192,  32, 160,  96, 224,  16, 144,
      80, 208,  48, 176, 112, 240,   8, 136,  72, 200,
      40, 168, 104, 232,  24, 152,  88, 216,  56, 184,
     120, 248,   4, 132,  68, 196,  36, 164, 100, 228,
      20, 148,  84, 212,  52, 180, 116, 244,  12, 140,
      76, 204,  44, 172, 108, 236,  28, 156,  92, 220,
      60, 188, 124, 252,   2, 130,  66, 194,  34, 162,
      98, 226,  18, 146,  82, 210,  50, 178, 114, 242,
      10, 138,  74, 202,  42, 170, 106, 234,  26, 154,
      90, 218,  58, 186, 122, 250,   6, 134,  70, 198,
      38, 166, 102, 230,  22, 150,  86, 214,  54, 182,
     118, 246,  14, 142,  78, 206,  46, 174, 110, 238,
      30, 158,  94, 222,  62, 190, 126, 254,   1, 129,
      65, 193,  33, 161,  97, 225,  17, 145,  81, 209,
      49, 177, 113, 241,   9, 137,  73, 201,  41, 169,
     105, 233,  25, 153,  89, 217,  57, 185, 121, 249,
       5, 133,  69, 197,  37, 165, 101, 229,  21, 149,
      85, 213,  53, 181, 117, 245,  13, 141,  77, 205,
      45, 173, 109, 237,  29, 157,  93, 221,  61, 189,
     125, 253,   3, 131,  67, 195,  35, 163,  99, 227,
      19, 147,  83, 211,  51, 179, 115, 243,  11, 139,
      75, 203,  43, 171, 107, 235,  27, 155,  91, 219,
      59, 187, 123, 251,   7, 135,  71, 199,  39, 167,
     103, 231,  23, 151,  87, 215,  55, 183, 119, 247,
      15, 143,  79, 207,  47, 175, 111, 239,  31, 159,
      95, 223,  63, 191, 127, 255);
-- end of specifications for 8bit reverse bits array

--------------------------------------------------------------------------------
--  declare the components
--------------------------------------------------------------------------------

COMPONENT HashAlgXbehv
  -- generic ();
  PORT (EnableIn : in EnableT;
        EnableOut : buffer EnableT;
        SymbolIn : in SymbolST;
        AddrOut : buffer TwelveBitST;
        Clock : in bit);
END COMPONENT;

COMPONENT HashMem
  -- generic ();
  PORT (Address : in TwelveBitST;
        Data : out TwelveBitST);
END COMPONENT;

COMPONENT Reg12
  -- generic ();
  PORT(Din : in bit_vector(11 downto 0);
       Dout : out bit_vector(11 downto 0);
       ClkIn : in bit);
END COMPONENT;

COMPONENT Mux12
  -- generic ();
  PORT(Din1 : in bit_vector(11 downto 0);
       Din2 : in bit_vector(11 downto 0);
       Dout : out bit_vector(11 downto 0);
       SelIn : in bit);
END COMPONENT;
```

```
  COMPONENT XorMulti
    -- generic ();
    PORT(Din1 : in bit_vector;
         Din2 : in bit_vector;
         Dout : out bit_vector);
  END COMPONENT;

  COMPONENT Reg1
    -- generic ();
    port(Din : in bit;
         Dout : out bit;
         ClkIn : in bit);
  END COMPONENT;

  COMPONENT AndGate
    -- generic ();
    port(Din1 : in bit;
         Din2 : in bit;
         Dout : out bit);
  END COMPONENT;

  COMPONENT InvGate
    -- generic ();
    port(Din : in bit;
         Dout : out bit);
  END COMPONENT;

End HashPKG;


-------------------------------------------------------------------------------
--  the body for package HashPKG
-------------------------------------------------------------------------------
PACKAGE body HashPKG IS

  -----------------------------------------------------------------------------
  --  procedure to fill arrays from a file
  -----------------------------------------------------------------------------
  PROCEDURE FillArray (VARIABLE ArrayName : out AddrArrayT;
                       VARIABLE FileName : in TEXT) is
    VARIABLE    counter          : natural := 0;
    VARIABLE    convert          : TwelveBitST;
    VARIABLE    line_buffer      : line;
  BEGIN
    readline(FileName, line_buffer);
    IF ((line_buffer'length = 0)) THEN
      ASSERT false
        REPORT "Bad file being used in FillArray procedure."
          SEVERITY failure;
    ELSE
      FileToArray : WHILE line_buffer'length /= 0 LOOP
        read(line_buffer, convert);
        ArrayName(counter) := convert;
        counter := counter + 1;
        readline(FileName, line_buffer);
      END LOOP FileToArray;
    END IF;
  end FillArray;
```

```
  ------------------------------------------------------------------------------
  --  function to XOR 2 integers
  ------------------------------------------------------------------------------
  FUNCTION IntXor (int1, int2, int1_2_len: in integer) RETURN integer is
    variable bit_int2, bit_xor, bit_int1: bit_vector((int1_2_len - 1) downto 0);
  begin
    bit_int2 := bit_vector_from_int(int2, bit_int2'length);
    bit_int1 := bit_vector_from_int(int1, bit_int1'length);
    bit_xor := bit_int1 xor bit_int2;
    return integer_from_bv(bit_xor);
  end IntXor;  -- end of function definition for integer xor function

  ------------------------------------------------------------------------------
  --  function to perform logical shift right on non-negative integer
  ------------------------------------------------------------------------------
  FUNCTION ShiftDown (value, size, count : in natural) RETURN natural is
    variable bit_value : bit_vector((size - 1) downto 0);
    variable bit_res : bit_vector(( (size - 1)   count) downto 0);
  BEGIN
    bit_value := bit_vector_from_int(value, bit_value'length);
    bit_res := bit_value( (size - 1) downto count);
    RETURN (integer_from_bv(bit_res));
  END ShiftDown;

  ------------------------------------------------------------------------------
  --  port conversion function, TwelveBitST to bit_vector
  ------------------------------------------------------------------------------
  FUNCTION TBtoBitConv (TBval : in TwelveBitST) RETURN bit_vector is
    --  unused declaration region
  BEGIN
    return(bit_vector_from_int(TBval, 12));
  END TBtoBitConv;

  ------------------------------------------------------------------------------
  --  port conversion function, bit_vector to TwelveBitST
  ------------------------------------------------------------------------------
  FUNCTION BitToTBConv (BVval : in bit_vector) RETURN TwelveBitST is
    --  unused declaration region
  BEGIN
    return(integer_from_bv(BVval));
  END BitToTBConv;

END HashPKG;
```

Appendix D4 - Source Code for dataconv_pkg.vhdl

This appendix contains the source code listing for the VHDL file dataconv_pkg.vhdl. The purpose of this package was to contain support subprograms needed by any of the other design units.

The information in this package was not contained in HashPKG.vhdl because this package was already in existence. The subprograms contained here could have been copied into HashPKG, but there was no need to do that. The building of code modules (design units) is one of the features of VHDL. The design unit simply needs to be analyzed into a library, the library made visible in the design unit needing the support functions, and all the capabilities are available. This allows the building of libraries which can allow even more rapid prototyping of future designs.

```
package dataconv_pkg is
function integer_from_bv (bv: in bit_vector) return integer;
function bit_vector_from_int (int,len: in integer) return bit_vector;
procedure int_to_bv (input: in integer;  output: out bit_vector);
procedure bv_to_int (input: in bit_vector;  output: out integer);
end dataconv_pkg;

package body dataconv_pkg is
procedure int_to_bv (input: in integer;  output: out bit_vector) is
  variable wkbuf  : integer := 0;
  begin
    -- positive value
    if input >= 0 then
       wkbuf := input;
       for i in output'reverse_range loop
    if (wkbuf mod 2) = 0 then
      output(i) := '0';
    else
      output(i) := '1';
    end if;
    wkbuf := wkbuf / 2;
       end loop;

     -- negative value
     else
       wkbuf := (-input) -1;
       for i in output'reverse_range loop
    if (wkbuf mod 2) = 0 then
      output(i) := '1';
    else
      output(i) := '0';
    end if;
      wkbuf := wkbuf / 2;
       end loop;

     end if;

   end int_to_bv;

procedure bv_to_int (input: in bit_vector;  output: out integer) is
  variable wkbuf : integer := 0;
  begin
    wkbuf := 0;
    for i in input'range loop
      wkbuf := wkbuf * 2;
      if input(i) = '1' then
   wkbuf := wkbuf + 1 ;
      end if;
    end loop;
    output := wkbuf;
  end bv_to_int;
```

```
function integer_from_bv(bv: in bit_vector) return integer is

  variable res : integer := 0;
  type bit_int_t is array (bit) of integer;
  constant bit_int : bit_int_t := (0,1);

  begin

    for i in bv'range loop
      res := res*2;
      res := res + bit_int(bv(i));
    end loop;
    return res;

  end integer_from_bv;


function bit_vector_from_int(int: in integer; len : in integer) return
bit_vector is

  variable res : bit_vector ( len-1 downto 0 );
  variable input: integer := int;
  type int_bit_t is array (0 to 1) of bit;
  constant int_bit : int_bit_t := ('0','1');

  begin

    assert int >= 0 report "negative integer converted to bit_vector";

    for i in 0 to len   1 loop
      res(i) := int_bit(input mod 2);
      input := input/2;
    end loop;

    assert input = 0 report "bits lost converting an integer to a bit_vector.";
    return res;

  end bit_vector_from_int;

end dataconv_pkg;
```

Appendix D5 - Source Code for HashConvert.vhdl

This appendix contains the source code listing for the VHDL file HashConvert.vhdl. The purpose of this program was to provide a transitional layer between the structural algorithm and the testbench. This was needed because the testbench was instantiating a component that used integer and enumerated types on its ports, while the structural component was using bits and bit_vectors. The reasons for this are explained in the main body of the text.

Supporting information is in HashPKG.vhdl. This contained the definitions of the types, subtypes, and subprograms needed by the algorithm. The package dataconv_pkg is also referenced because it contains the conversion subprograms to and from bit_vector and integer.

```
--------------------------------------------------------------------------------
--  HashConvert.vhdl
--  This file will hold the vhdl to convert the ports on the hash algorithm
--  entity from natural to bit_vector, and vice versa.
--------------------------------------------------------------------------------

USE work.HashPkg.all;
USE work.dataconv_pkg.all;

--------------------------------------------------------------------------------
--  entity for the conversion
--------------------------------------------------------------------------------
ENTITY HashConvert IS
  port( EnableIn  : in EnableT;
        EnableOut : buffer EnableT;
        SymbolIn  : in SymbolST;
        AddrOut   : buffer TwelveBitST;
        Clock     : in bit);
END HashConvert;

--------------------------------------------------------------------------------
--  architecture for the conversion
--------------------------------------------------------------------------------
ARCHITECTURE HashConvertArch OF HashConvert IS
  SIGNAL        BitSymbol          : bit_vector(7 downto 0);
  SIGNAL        BitAddr            : bit_vector(11 downto 0);
  SIGNAL        EnableInBitConv    : bit;
  SIGNAL        EnableOutConv      : bit;

  COMPONENT HashAlgXstru
    -- generic ();
    PORT (EnableInBit : in bit;
          EnableOutBit : buffer bit;
          SymbolInBit : in bit_vector(7 downto 0);
          AddrOutBit : buffer bit_vector(11 downto 0);
          Clock : in bit);
  END COMPONENT;
Begin  --  HashAlgXstruArch

  BitSymbol <= bit_vector_from_int(SymbolIn, BitSymbol'length);
  AddrOut <= integer_from_bv(BitAddr);

  WITH EnableIn SELECT
    EnableInBitConv <=
    '0' WHEN stop,
    '1' WHEN run
    ;

  WITH EnableOutConv SELECT
    EnableOut <=
    stop WHEN '0',
    run  WHEN '1'
    ;

  UXSTR : HashAlgXstru
    port map(EnableInBit => EnableInBitConv,
             EnableOutBit => EnableOutConv,
             SymbolInBit => BitSymbol,
             AddrOutBit => BitAddr,
             Clock => Clock);

END HashConvertArch;
```

Appendix D6 - Source Code for HashAlgXstru.vhdl

This appendix contains the source code listing for the VHDL file HashAlgXstru.vhdl. The purpose of this program was to model the XOR hashing algorithm as a structural VHDL model. All of the stimulus was provided by the testbench, and all of the results were returned to the testbench.

Supporting information is in HashPKG.vhdl. This contained the definitions of the types, subtypes, and subprograms needed by the algorithm. To do the calculations, data representing the previous codes was needed. This was accessed through the component called HashMem. The rest of the algorithm is performed by the other components, and by the interconnection between them.

```
--------------------------------------------------------------------------------
--  HashAlgXstru.vhdl
--  This file will hold the vhdl to implement the structural version of the
--  XOR hashing algorithm.  It will accept data from the test bench, generate
--  the address using a memory component and send the address back to the test
--  bench.
--------------------------------------------------------------------------------

USE work.HashPkg.all;

--------------------------------------------------------------------------------
--  entity for the structural version of the XOR hashing algorithm
--------------------------------------------------------------------------------
ENTITY HashAlgXstru IS
  port( EnableInBit  : in bit;
        EnableOutBit : buffer bit;
        SymbolInBit  : in bit_vector(7 downto 0);
        AddrOutBit   : buffer bit_vector(11 downto 0);
        Clock        : in bit);
  BEGIN
    ASSERT false
      REPORT "Executing the structural version of the algorithm."
        SEVERITY note;
END HashAlgXstru;

--------------------------------------------------------------------------------
--  architecture for the structural version of the XOR hashing algorithm
--------------------------------------------------------------------------------
ARCHITECTURE HashAlgXstruArch OF HashAlgXstru IS
  SIGNAL        PcodeVal        : bit_vector(11 downto 0) := X"FFF";
  SIGNAL        MPcode          : bit_vector(11 downto 0) := X"F0F";
  SIGNAL        Xaddr           : bit_vector(11 downto 0);
  SIGNAL        Taddr           : bit_vector(11 downto 0);
  SIGNAL        MemVal          : bit_vector(11 downto 0);
  SIGNAL        First           : bit := '1';
  SIGNAL        MuxSel          : bit := '0';
  SIGNAL        Zero4           : bit_vector(3 downto 0) := X"0";
  SIGNAL        Sym4X           : bit_vector(3 downto 0);
  SIGNAL        AndOut          : bit;
  SIGNAL        NClock          : bit;
  SIGNAL        OutOK           : bit;
  SIGNAL        OutAnd          : bit;
  SIGNAL        TenOutBit       : bit;
Begin  --  HashAlgXstruArch

  --------------------------------------------------------------------------------
  --  instantiate the components
  --------------------------------------------------------------------------------
  UENINREG : Reg1
    port map(Din => AndOut,
             Dout => OutOK,
             ClkIn => NClock);

  UENINAND : AndGate
    port map(Din1 => EnableInBit,
             Din2 => First,
             Dout => AndOut);

  UNCLK : InvGate
    port map(Din => Clock,
             Dout => NClock);

  EnableOutBit <= TenOutBit;              --  can't port map out to buffer
  UENOUTREG : Reg1
    port map(Din => OutAnd,
             Dout => TenOutBit,
             ClkIn => Clock);
```

```
  UENOUTAND : AndGate
    port map(Din1 => EnableInBit,
             Din2 => OutOK,
             Dout => OutAnd);

  UMEM : HashMem
    port map(Address => BitToTBConv(AddrOutBit),
             TBtoBitConv(Data) => MemVal);

  UMUX : Mux12
    port map(Din1(11 downto 8) => Zero4,
             Din1(7 downto 0) => SymbolInBit,
             Din2 => MemVal,
             Dout => MPcode,
             SelIn => EnableOutBit);

  UPREG : Reg12
    port map(Din => MPcode,
             Dout => PcodeVal,
             ClkIn => NOT Clock);

  AddrOutBit <= Taddr;                    --  can't port map out to buffer
  UAREG : Reg12
    port map(Din => Xaddr,
             Dout => Taddr,
             ClkIn => Clock);


  UXSYM : XorMulti
    port map(Din1(0) => SymbolInBit(0),
             Din1(1) => SymbolInBit(1),
             Din1(2) => SymbolInBit(2),
             Din1(3) => SymbolInBit(3),
             Din2(0) => SymbolInBit(7),
             Din2(1) => SymbolInBit(6),
             Din2(2) => SymbolInBit(5),
             Din2(3) => SymbolInBit(4),
             Dout => Sym4X);

  UXADDR : XorMulti
    port map(Din1 => PcodeVal,
             Din2(0) => PcodeVal(8),
             Din2(1) => PcodeVal(9),
             Din2(2) => PcodeVal(10),
             Din2(3) => PcodeVal(11),
             Din2(4) => Sym4X(0),
             Din2(5) => Sym4X(1),
             Din2(6) => Sym4X(2),
             Din2(7) => Sym4X(3),
             Din2(8) => Sym4X(3),
             Din2(9) => Sym4X(2),
             Din2(10) => Sym4X(1),
             Din2(11) => Sym4X(0),
             Dout => Xaddr);

END HashAlgXstruArch;
```

Appendix D7 - Source Code for HashMem.vhdl

This appendix contains the source code listing for the VHDL file HashMem.vhdl. The purpose of this program was to represent a read only memory. The data needed for the memory was read from a file opened by the component. The file was opened and read on the first access (during elaboration). The data read was stored in an array, and the accesses used the data from the array.

Supporting information is in HashPKG.vhdl. This contained the definitions of the types, subtypes, and subprograms needed by the algorithm. The textio package is also referenced because the file is opened as a file of text.

INPUTS a file representing the previous codes.

```
USE work.HashPkg.all;
USE std.textio.all;

-------------------------------------------------------------------------------
--  entity to emulate the memory that stores the previous codes
-------------------------------------------------------------------------------
ENTITY HashMem IS
  port( Address : in TwelveBitST;
        Data : out TwelveBitST);
END HashMem;


-------------------------------------------------------------------------------
--  architecture for the memory element
-------------------------------------------------------------------------------
ARCHITECTURE HashMemArch OF HashMem IS
  FILE          PcodeFile           : TEXT is in "PcodeFile";
BEGIN  --  HashMemArch
  -----------------------------------------------------------------------------
  --  Process:    Memory
  --  Purpose:    To accept an address, read the data from the array (the
  --              memory), and to return the new data.
  --              NOTE:  This process will work two very different ways
  --              depending on how PcodeArray is declared.  If it is a signal,
  --              then when it gets initialized at elaboration time the values
  --              will not actually become available until after a simulation
  --              delta.  This means that Data (the signal) will not get the
  --              "new" value referenced by Address, it will get the "old"
  --              value which is 0 (because the array gets all zeros upon
  --              creation).  BUT, if PcodeArray is declared a variable then
  --              the values become available immediatly and Data will get
  --              the "new" value referenced by Address.
  -----------------------------------------------------------------------------
  Memory : PROCESS (Address)
    VARIABLE      First           : boolean := true;
    VARIABLE        PcodeArray          : AddrArrayT(TwelveBitST);
  BEGIN  --  PROCESS Memory
    IF ( First ) THEN
      FillArray(PcodeArray, PcodeFile);
      First := false;
      Data <= PcodeArray(Address) after (HalfPeriod/2);
    ELSE
      Data <= PcodeArray(Address) after (HalfPeriod/2);
                                          --  Address
    END IF;
  END PROCESS Memory;
END HashMemArch;
```

Appendix D8 - Source Code for Reg12.vhdl

This appendix contains the source code listing for the VHDL file Reg12.vhdl The purpose of this program was to represent a 12 bit wide synchronous register. It used a delay on the signal assignment for the output data to make simulation and debugging easier. It can be difficult to determine the sequence of events if only delta delays are used.

```
--------------------------------------------------------------------------------
--  This file will contain the VHDL to simulate a 12 bit register for the
--  structural description.
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--  entity for the 12 bit register
--------------------------------------------------------------------------------
ENTITY Reg12 IS
  port(Din : in bit_vector(11 downto 0);
       Dout : out bit_vector(11 downto 0);
       ClkIn : in bit);
END Reg12;

--------------------------------------------------------------------------------
--  architecture for the 12 bit register
--------------------------------------------------------------------------------
ARCHITECTURE Reg12Arch OF Reg12 IS
  --  unused declaration area
BEGIN  --  Reg12Arch
  ------------------------------------------------------------------------------
  --  Process:    Reg
  --  Purpose:    A simulation of a rising edge sensitive 12 bit wide register.
  ------------------------------------------------------------------------------
  Reg : PROCESS (ClkIn)
    --  unused declaration area
  BEGIN  --  PROCESS Reg
    IF (ClkIn = '1') THEN
      Dout <= Din after 3 ns;            --  delay to ease debugging
    END IF;
  END PROCESS Reg;
END Reg12Arch;
```

Appendix D9 - Source Code for Mux12.vhdl

This appendix contains the source code listing for the VHDL file Mux12.vhdl The purpose of this program was to represent a 12 bit wide, two to one multiplexor. It used a delay on the signal assignment for the output data to make simulation and debugging easier. It can be difficult to determine the sequence of events if only delta delays are used.

```
--------------------------------------------------------------------------------
--  This file will contain the VHDL to simulate a 12 bit multiplexer for the
--  structural description.  This is a two to one mux.
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--  entity for the 12 bit multiplexer
--------------------------------------------------------------------------------
ENTITY Mux12 IS
  port(Din1 : in bit_vector(11 downto 0);
       Din2 : in bit_vector(11 downto 0);
       Dout : out bit_vector(11 downto 0);
       SelIn : in bit);
END Mux12;

--------------------------------------------------------------------------------
--  architecture for the 12 bit multiplexer
--------------------------------------------------------------------------------
ARCHITECTURE Mux12Arch OF Mux12 IS
  --  unused declaration area
BEGIN  --  Mux12Arch
  WITH SelIn SELECT
    Dout <=
    Din1 after 4 ns WHEN '0',     --  delay to ease debugging
    Din2 after 4 ns WHEN '1'      --  delay to ease debugging
    ;
END Mux12Arch;
```

Appendix D10 - Source Code for XorMulti12.vhdl

This appendix contains the source code listing for the VHDL file XorMulti12.vhdl. The purpose of this program was to represent a variable width device that consisted of two input XOR gates. Unconstrained arrays were used for the inputs so that this component could represent any number of XOR devices, and in fact can represent a different number of devices each time it is instantiated. This was accomplished through the use of the generate statement. The instantiation process can be seen in HashAlgXstru.vhdl. The component uses a delay on the signal assignment for the output data to make simulation and debugging easier. It can be difficult to determine the sequence of events if only delta delays are used.

PROGRAM PARAMETERS

program name
first argument how many output values to produce

```
--------------------------------------------------------------------------------
--  This file will contain the VHDL to simulate a unconstrained exclusive or
--  for the structural description.
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--  entity for the exclusive or
--------------------------------------------------------------------------------
ENTITY XorMulti IS
  port(Din1 : in bit_vector;
       Din2 : in bit_vector;
       Dout : out bit_vector);
  BEGIN
    ASSERT Din1'length = Din2'length
      REPORT "Different length input arrays to XOR."
        SEVERITY failure;
    ASSERT Din1'length = Dout'length
      REPORT "Different lenght input and output arrays to XOR."
        SEVERITY failure;
END XorMulti;

--------------------------------------------------------------------------------
--  architecture for the exclusive or
--  Note the use of unconstrained arrays on the ports, and the generate
--  statement which gets its index from the length of the input array.
--------------------------------------------------------------------------------
ARCHITECTURE XorMultiArch OF XorMulti IS
BEGIN  --  XorMultiArch
  Gen1: for I in ((Din1'length)   1) downto 0 GENERATE
  Dout(I) <= Din1(I) XOR Din2(I) after 2 ns;--  delay to ease debugging
  END GENERATE Gen1;
END XorMultiArch;
```

Appendix D11 - Source Code for MiscComp.vhdl

This appendix contains the source code listing for the VHDL file MiscComp.vhdl. The purpose of this file was to contain all the other miscellaneous components needed to complete the structural description of the algorithm. Since these are all simple components, and because VHDL style allows it, all the components were put in the same file for simplicity.

This file contains the following devices: Reg1, a single bit wide synchronous register, AndGate, a two input logical and gate, and InvGate, a single input logical inversion. The register could have been constructed using the generate statement since other width devices of a similar structure are used, but this method was chosen to illustrate a different modeling technique. The components all use a delay on the signal assignment for the output data to make simulation and debugging easier. It can be difficult to determine the sequence of events if only delta delays are used.

```
--------------------------------------------------------------------------------
--  This file will contain the VHDL to simulate miscellaneous components for
--  the structural description.
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--  entity for a single bit register
--------------------------------------------------------------------------------
ENTITY Reg1 IS
  port(Din : in bit;
       Dout : out bit;
       ClkIn : in bit);
END Reg1;

--------------------------------------------------------------------------------
--  architecture for a single bit register
--------------------------------------------------------------------------------
ARCHITECTURE Reg1Arch OF Reg1 IS
  --  unused declaration area
BEGIN  --  Reg1Arch
  ------------------------------------------------------------------------------
  --  Process:   Reg
  --  Purpose:   A simulation of a rising edge sensitive single bit register.
  ------------------------------------------------------------------------------
  Reg : PROCESS (ClkIn)
    --  unused declaration area
  BEGIN  --  PROCESS Reg
    IF (ClkIn = '1') THEN
      Dout <= Din after 3 ns;            --  delay to ease debugging
    END IF;
  END PROCESS Reg;
END Reg1Arch;

--------------------------------------------------------------------------------
--  entity for a AND gate
--------------------------------------------------------------------------------
ENTITY AndGate IS
  port(Din1 : in bit;
       Din2 : in bit;
       Dout : out bit);
END AndGate;

--------------------------------------------------------------------------------
--  architecture for a AND gate
--------------------------------------------------------------------------------
ARCHITECTURE AndGateArch OF AndGate IS
  --  unused declaration area
BEGIN  --  AndGateArch
  Dout <= Din1 AND Din2 after 1 ns;      --  delay to ease debugging
END AndGateArch;

--------------------------------------------------------------------------------
--  entity for and Inverter
--------------------------------------------------------------------------------
ENTITY InvGate IS
  port(Din : in bit;
       Dout : out bit);
END InvGate;

--------------------------------------------------------------------------------
--  architecture for an Inverter
--------------------------------------------------------------------------------
ARCHITECTURE InvGateArch OF InvGate IS
  --  unused declaration area
BEGIN  --  InvGateArch
  Dout <= NOT Din after 2 ns;            --  delay to ease debugging
END InvGateArch;
```

BIBLIOGRAPHY

Ashenden, Peter J., *The VHDL Cookbook*, First Edition, Department of Computer Science, University of Adelaide, South Australia, July 1990.

Bailey, R. L., Mukkamala, R., *Pipelining Data Compression Algorithms*, The Computer Journal, Volume 33, Number 4, 1990.

Barnes, J. G. P., *Programming in ADA*, Third Edition, Addison Wesley, 1989.

Chang, C. C., Chen, C. Y., Jan, J. K., *On the Design of a Machine-Independent Perfect Hashing Scheme*, The Computer Journal, Volume 34, Number 5, 1991, pp. 469-474.

Coelho, David R., *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

Fox, Jeffrey R., *A higher level of synthesis*, IEEE Spectrum, Volume 30, Number 3, March 1993, page 43.

Graham, Neill, *Microprocessor Programming for Computer Hobbyists*, Tab Books, 1977.

Harr, Randolph E., *The Applications of VHDL to Circuit Design*, Kluwer Academic Publishers, 1991.

Hennessey, John L., Patterson, Davis A., *Computer Architecture, A Quantative Approach*, Morgan Kaufmann Publishers, Inc. 1990.

IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076-1987, Institute of Electrical and Electronic's Engineers Inc., 1988.

Lipsett, Roger; Schaefer, Carl; Ussery, Cary; *VHDL. Hardware Description and Design*, Kluwer Academic Publishers, 1989.

Knuth, Donald E., *The Art of Computer Programming*, Second Edition, Addison-Wesley, 1973.

Mathcad User's Guide, Windows Version, Mathsoft, Inc., 1992.

Meyer, Ernest, *The Basics of Logic Synthesis*, ASIC & EDA, February 1993, pp 10-15.

Perry, Douglas L., *VHDL*, McGraw-Hill, Inc., 1991.

Rosen, Kenneth H., *Discrete mathematics and its Applications*, Random House, 1988.

Vantage Spreadsheet User Guide, Vantage Analysis Systems, 1992.